ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG
INSTITUT FÜR INFORMATIK

Lehrstuhl für Rechnerarchitektur
Prof. Dr. Bernd Becker

k-Induktion am Beispiel des Intervall-Constraint-Solvers iSAT

Bachelorarbeit

Leonore Winterer
Betreuer: Stefan Kupferschmid
12. März 2012

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Außerdem erkläre ich, dass diese Bachelorarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Leonore Winterer
Freiburg im *März 2012*.

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen **3**
2.1 Verifikation 3
2.2 Aussagenlogik 5
2.3 Schaltkreise 8
2.4 Modelle 10
2.5 Eigenschaften 12
2.6 Das Erfüllbarkeitsproblem der Aussagenlogik 14
2.7 Bounded Model Checking 16

3 iSAT **17**
3.1 SAT modulo Theory 17
3.2 iSAT Syntax 18
3.3 Intervall-Arithmetik 21
3.4 Inkrementelles Lösen 22

4 k-Induktion **23**
4.1 Einfacher Ansatz 23
4.2 Erweiterter Ansatz 23
4.3 Uniqueness 25
4.4 k-Induktion für arithmetische Transitionssysteme 28

5 Implementierung **28**
5.1 Naive Implementierung 29
5.2 Implementierung mit teurer Uniqueness 30
5.3 Dynamische Uniqueness 31

6 Ergebnisse **32**
6.1 Mächtigkeit 33
6.2 Performanz 35

7 Zusammenfassung und Ausblick **40**

Literaturverzeichnis **41**

1 Einleitung

Diese Arbeit beschäftigt sich mit der k-Induktion, einem unter anderem in [1] vorgestellten Ansatz zur Erweiterung von Bounded Model Checking. Dabei handelt es sich um eine Technik, die verwendet werden kann, um schrittweise die Korrektheit eines Systems zu überprüfen. Ursprünglich war es dadurch lediglich möglich, mithilfe von Gegenbeispielen Fehler im System aufzuzeigen, durch die Kombination mit k-Induktion wird es jedoch ermöglicht, auch die Abwesenheit von Fehlern zu beweisen.
Die einzelnen Kapitel der Arbeit führen zunächst einige für das weitere Verständnis notwendige Grundlagen ein. Der SMT-Solver iSAT, der im Zuge dieser Arbeit um eine Implementierung der k-Induktion erweitert wurde, wird vorgestellt. Anschließend wird das Prinzip der k-Induktion eingeführt, ihre Korrektheit und Vollständigkeit werden bewiesen. Zu guter Letzt werden die eigentliche Implementierung sowie die beim Lösen einer großen Menge boolscher Benchmarks erzielten Ergebnisse diskutiert.

2 Grundlagen

Dieses Kapitel soll einige grundlegende Begriffe erläutern, die zum Verständnis der folgenden Kapitel benötigt werden. Die einzelnen Abschnitte führen Schreibweisen, Definitionen und Begriffe ein, deren Kenntnis im späteren Verlauf der Arbeit vorausgesetzt wird.

2.1 Verifikation

Informations- und kommunikationstechnische Systeme halten immer mehr in unserem täglichen Leben Einzug. Während Onlinebanking und GPS-Tracking noch vor wenigen Jahren nur von einzelnen Menschen genutzt wurden, erachten wir diese inzwischen als selbstverständlich. Je mehr wir uns im Alltag auf dererlei *eingebettete Systeme* verlassen, desto abhängiger werden wir auch von ihrer korrekten Funktionsweise. Es gewinnt also mehr und mehr an Bedeutung, sicherzustellen, dass all diese Systeme zuverlässig und fehlerfrei arbeiten.
Deshalb muss bereits beim Design dieser Systeme darauf geachtet werden, dass das Endprodukt möglichst exakt der Spezifikation entspricht. Unter einer *Spezifikation* versteht man dabei eine (formale oder informale) Beschreibung aller gewünschten Eigenschaften eines Systems (siehe auch Abschnitt 2.5). Um diese Übereinstimmung zwischen Spezifikation und Endprodukt zu gewährleisten, stehen unterschiedliche Techniken zur Verfügung, die unter dem Begriff *Verifikation* zusammengefasst werden. Gemeinsames Ziel aller

Verifikationsmethoden ist es, festzustellen, ob das entwickelte System gemäß seiner Spezifikation korrekt funktioniert (das System ist *sicher* bezgl. der Spezifikation), oder Fehler aufweist. Im Fall eines Fehlers sollte die Technik möglichst einen Hinweis darauf liefern, welcher Teil des Systems fehlerhaft ist oder wie er behoben werden kann.
Neben informellen Techniken wie dem Testen eines Systems oder der händischen Fehlersuche durch die Entwickler gibt es auch formale Techniken, die die Korrektheit eines Systems beweisen können. Dazu gehört auch das sog. *Model Checking*, das im folgenden Abschnitt behandelt wird.

Model Checking

Wie eingangs erwähnt ist die Verifikation von Hard- und Softwaresystemen eine Disziplin, die in der heutigen Zeit zunehmend wichtiger wird. Besondere Bedeutung kommt dabei der Technik des *Model Checkings* zu.

Definition 1 (Model Checking)**.**
Model Checking ist eine automatisierte Technik, die – gegeben ein endliches Modell eines Systems und eine formale Eigenschaft – systematisch überprüft, ob die Eigenschaft für dieses Modell (oder einen seiner Zustände) gültig ist.
Nach [2, S.11]

Ausgangspunkt des Verfahrens sind also ein zu verifizierendes System und eine Eigenschaft, die dieses System erfüllen soll. Aus dem gegebenen System wird ein Modell erstellt, das das Verhalten des Systems möglichst korrekt und eindeutig wiedergibt. Oft ist das erzeugte Modell ein *Transitionssystem* mit bestimmten Eigenschaften (siehe Abschnitt 2.4).
Die zu überprüfende Eigenschaft muss ebenfalls formalisiert werden. Dazu stehen verschiedene logische Spezifikationssprachen zur Verfügung (siehe auch Abschnitt 2.5).
Nun kann ein speziells Programm, der sog. *Modelchecker*, mit dem eigentlichen Verifikationsprozess beginnen, der meist automatisiert abläuft. Als Ergebnis erhält man entweder die Bestätigung, dass die Eigenschaft in dem Modell gültig ist, oder ein *Gegenbeispiel*, also ein Verhalten des Systems, das die Eigenschaft verletzt. Wird dieses Gegenbeispiel im Modell analysiert, ist es möglich, den Ort des Fehlers zu finden, und das System und sein Modell entsprechend anzupassen. Halten alle in der Spezifikation genannten Eigenschaften im Modell, so ist das System korrekt bezüglich seiner Spezifikation.
Der eben beschriebene Prozess wird in Abbildung 1 veranschaulicht.
Im nächsten Abschnitt wird die Aussagenlogik eingeführt, die eine wichtige Grundlage für viele Techniken und Algorithmen des Model Checkings bildet.

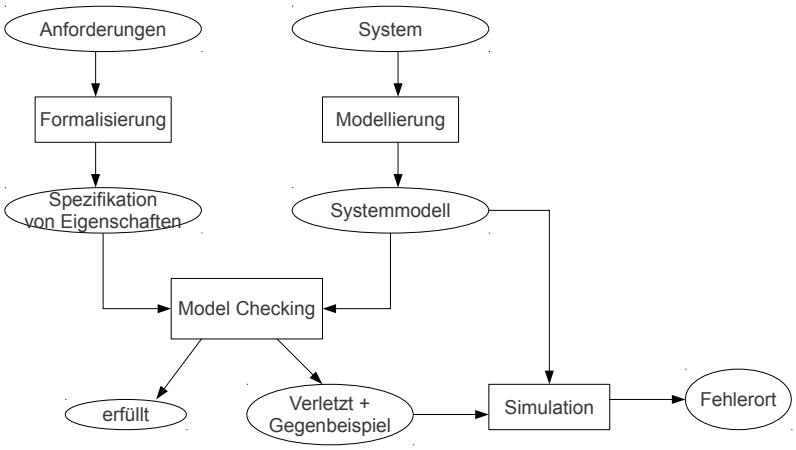

Abbildung 1: Schema des Modelchecking-Ansatzes nach [2, S.8]

2.2 Aussagenlogik

In diesem Abschnitt soll formal die *Aussagenlogik* eingeführt werden. In diesem Zusammenhang werden eine Reihe wichtiger Begriffe und Notationen vorgestellt. Die Aussagenlogik ist ein Teilgebiet der mathematischen Logik, die sich mit *Aussagen* beschäftigt. Eine Aussage kann *wahr* oder *falsch* sein. Analog zu [3] wird die Syntax der Aussagenlogik folgendermaßen definiert werden:

Definition 2 (Syntax der Aussagenlogik)**.**

a) *Atomare Formeln* sind ausschließlich einfache Aussagen der Form x_i mit $i \in \mathbb{N}$.

b) Aussagenlogische Formeln werden ausschließlich induktiv durch Anwendung der folgenden Schritte definiert:

   1) Jede atomare Formel ist eine Formel.
   2) Sind F_1 und F_2 Formeln, so sind auch $(F_1 \vee F_2)$ und $(F_1 \wedge F_2)$ Formeln.
   3) Für jede Formel F ist auch $\neg F$ eine Formel.

Man bezeichnet $(F_1 \wedge F_2)$ auch als *Konjunktion* und $(F_1 \vee F_2)$ als *Disjunktion* von F_1 und F_2. $\neg F$ heißt *Negation* von F. Atomare Formeln werden auch als *Variablen* bezeichnet. Eine Variable oder die Negation einer Variable nennt man auch positives bzw. negatives *Literal*. Folgende Notations-Konventionen sollen die Formulierung langer Formeln einfacher gestalten:

Notation 1.
Seien Formeln F_1, F_2, F_3,... gegeben, so schreibt man:

a) $(\neg F_1 \vee F_2)$ als $(F_1 \Rightarrow F_2)$ (*Implikation*),

b) $((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$ als $(F_1 \Leftrightarrow F_2)$ (*Äquivalenz*),

c) $(\ldots((F_1 \vee F_2) \vee F_3) \vee \ldots)$ als $(F_1 \vee F_2 \vee F_3 \vee \ldots)$ (analog für $\wedge$).

Außerdem kann das jeweils äußerste Klammernpaar einer Formel entfernt werden, wenn aus dem Zusammenhang ersichtlich ist, dass es sich um eine Formel handelt.

Den mithilfe dieser Regeln definierten Formeln muss jetzt noch ein *Wahrheitswert* zugeordnet werden. Wahrheitswerte sind die Elemente der Menge $\{wahr, falsch\}$. Üblicherweise wird diese Menge jedoch durch die boolsche Wertemenge $\{0, 1\}$ ersetzt, wobei 1 für *wahr* steht und 0 für *falsch*. Die Zuordnung der Wahrheitswerte zu den Formeln erfolgt nach folgenden Regeln:

Definition 3 (Semantik der Aussagenlogik)**.** Sei M eine Menge von atomaren Formeln. Eine *Belegung* ist eine Abbildung $\alpha : M \to \{0, 1\}$, die jeder atomaren Formel einen Wahrheitswert zuordnet. Für die Menge $\hat{M}$ von beliebigen Formeln, die über den atomaren Formeln aus M gebildet wurden, kann α zu einer Belegung $\hat{\alpha} : \hat{M} \to \{0, 1\}$ folgendermaßen erweitert werden:

- $\hat{\alpha}(x) = \alpha(x)$ für alle $x \in M$
- $\hat{\alpha}((F_1 \wedge F_2)) = 1$ gdw. $\hat{\alpha}(F_1) = 1$ und $\hat{\alpha}(F_2) = 1$, für $F_1, F_2 \in \hat{M}$
- $\hat{\alpha}((F_1 \vee F_2)) = 1$ gdw. $\hat{\alpha}(F_1) = 1$ oder $\hat{\alpha}(F_2) = 1$, für $F_1, F_2 \in \hat{M}$
- $\hat{\alpha}(\neg F) = 1$ gdw. $\hat{\alpha}(F) = 0$ für $F \in \hat{M}$.

Existiert zu einer Formel F eine *passende* Belegung α (die allen in F vorkommenden atomaren Aussagen einen Wahrheitswert zuweist), so dass $\hat{\alpha}(F) = 1$ gilt, so ist F *erfüllbar*. Man nennt α ein *Modell* von F und schreibt $\alpha \models F$. Exisitiert zu einer Formel F kein Modell α, so ist F *unerfüllbar*.
Zwei aussagenlogische Formeln F_1 und F_2 heißen äquivalent $(F_1 \equiv F_2)$, wenn für jede Belegung α gilt: $\alpha \models F_1$ gdw. $\alpha \models F_2$.

Da für jede endliche Menge M von atomaren Formeln nur endlich viele Abbildungen $\alpha : M \to \{0, 1\}$ existieren, kann die Äquivalenz einfacher Formeln leicht durch Anwendung der Regeln aus Definition 3 überprüft werden. Dabei stößt man unter anderem auf folgende Äquivalenzen:

Satz 1 (Äquivalenzen der Aussagenlogik)**.** Für alle aussagenlogischen Formeln F_1, F_2, F_3 gilt:

- $(F_1 \wedge F_1) \equiv F_1$ und $(F_1 \vee F_1) \equiv F_1$ (Idempotenz)
- $(F_1 \wedge F_2) \equiv (F_2 \wedge F_1)$ und $(F_1 \vee F_2) \equiv (F_2 \vee F_1)$ (Kommutativität)
- $\neg\neg F_1 \equiv F_1$ (Doppelnegation)
- $(F_1 \wedge (F_2 \vee F_3)) \equiv ((F_1 \wedge F_2) \vee (F_1 \wedge F_3))$ und $(F_1 \vee (F_2 \wedge F_3)) \equiv ((F_1 \vee F_2) \wedge (F_1 \vee F_3))$ (Distributivität)
- $\neg(F_1 \wedge F_2) \equiv (\neg F_1 \vee \neg F_2)$ und $\neg(F_1 \vee F_2) \equiv (\neg F_1 \wedge \neg F_2)$ (Regel von de Morgan)

Mithilfe dieser Äquivalenzen können aussagenlogische Formeln umgeformt werden, etwa um sie in eine bestimmte Form zu bringen. Eine im Rahmen dieser Arbeit wichtige Form ist die *Konjunktive Normalform* (*KNF*), die als Eingabeformat für die sog. *SAT-Solver* fungiert (siehe Abschnitt 2.6).

Definition 4 (Konjunktive Normalform)**.**
Eine Formel F ist in KNF, wenn sie eine Konjunktion von Disjunktionen von Literalen ist, es also Literale L_{ij} gibt, so dass
$F = (L_{11} \vee \cdots \vee L_{1m_1}) \wedge \cdots \wedge (L_{n1} \vee \cdots \vee L_{nm_n})$
[3, S.27]

Die Teilformeln der Form $(L_{11} \vee \cdots \vee L_{1m_1})$ werden *Klauseln* genannt. Aus Definition 3 folgt: Eine Belegung α erfüllt eine aussagenlogische Formel in KNF F genau dann, wenn jede Klausel von F durch α erfüllt wird. Eine Klausel wird genau dann von α erfüllt, wenn mindestens eines ihrer Literale von α erfüllt wird.
Mithilfe der oben erwähnten Äquivalenzen kann jede beliebige aussagenlogische Formel in eine KNF umgeformt werden. Diese Umformung erzeugt im schlimmsten Fall jedoch eine exponentiell längere Formel[1]. Mit der *Tseitin-Transformation*[2] existiert eine Methode, eine beliebige aussagenlogische Formel in eine maximal linear längere, *erfüllbarkeitsäquivalente* KNF umzuwandeln. Erfüllbarkeitsäquivalent sind zwei Formeln F_1 und F_2 genau dann, wenn

[1] Die *Länge* einer Formel bezieht sich dabei auf die Anzahl der enthaltenen Operatoren.
[2] Siehe [4].

gilt: $\exists M_1 \models F_2$ gdw. $\exists M_2 \models F_2$.
Auf die Funktionsweise der Tseitin-Transformation soll an dieser Stelle nicht weiter eingegangen werden.
Der nächste Abschnitt beschäftigt sich mit der Logik von *elektronischen Schaltkreisen*, die mithilfe der Aussagenlogik modelliert werden kann.

2.3 Schaltkreise

Als *Schaltkreis* oder auch Schaltnetz bezeichnet man ein Netz aus elementaren logischen *Gattern*. Diese Gatter wiederum implementieren elementare logische Operationen, wie beispielsweise das *AND*-Gatter (Konjunktion), das *OR*-Gatter (Disjunktion) und das *NOT*-Gatter (Negation). Gatter werden technisch durch Transistoren realisiert. Für mehr Details zur technischen Realisierung siehe auch [5].
Schaltkreise korrespondieren zu boolschen Funktionen – jede boolsche Funktion lässt sich als Schaltkreis darstellen, und jedem Schaltkreis liegt eine boolsche Funktion zu Grunde. Dabei ist eine boolsche Funktion folgendermaßen definiert:

Definition 5 (Boolsche Funktion)**.**
Eine boolsche Funktion ist eine Abbildung $\{0,1\}^n \to \{0,1\}^m$. Dabei bezeichnet man n als die Anzahl der *Eingänge* und m als die Anzahl der *Ausgänge* der Funktion.

Jede boolsche Funktion kann durch m aussagenlogische Formeln dargestellt werden, die über n gemeinsamen Variablen definiert sind. Die Eingänge der Funktion entsprechen dabei den Variablen der aussagenlogischen Formeln, die Ausgänge ihrem unter Anwendung der Regeln aus Definition 3 evaluierten Wahrheitswert. Dieser Zusammenhang zwischen boolschen Funktionen und aussagenlogischen Formeln soll an dieser Stelle als gegeben hingenommen werden. Den Beweis dafür kann beispielsweise in [5, Kapitel 2] nachgelesen werden.
Graphisch können Schaltkreise als azyklische, gerichtete Graphen repräsentiert werden.

Definition 6 (Gerichteter Graph)**.** $G = (V, E)$ heißt gerichteter Graph, wenn gilt:

- V ist eine endliche, nicht-leere Menge von *Knoten*
- $E \subseteq (V \times V)$ ist eine endliche Menge von *Kanten*

- Auf E sind Abbildungen $Q : E \to V$ und $Z : E \to V$ definiert, die jeder Kante eine *Quelle* und ein *Ziel* zuordnen. So wird genau festgelegt, von welchem Knoten eine Kante ausgeht und wohin sie führt.

Nach [5, Definition 6.1]

Eine Folge $l_1, l_2, \ldots, l_n$ mit $l_i \in E$ für alle i heißt *Pfad* der Länge n, wenn für alle $1 \leq i \leq n-1$ $Z(l_i) = Q(l_{i+1})$ gilt. Ein gerichteter Graph $G = (V, E)$ heißt *azyklisch*, wenn es darin keinen Pfad $l_1, l_2, \ldots, l_n$ gibt mit $Z(l_n) = Q(l_1)$. Der Schaltkreis in Abbildung 2 ist ein Beispiel für einen solchen azyklischen, gerichteten Graphen.

Kombinatorische Schaltkreise

Schaltkreise, die neben den Ein- und Ausgängen lediglich aussagenlogische Operatoren verwenden, bezeichnet man auch als *kombinatorische* Schaltkreise. Um eine aussagenlogische Formel als kombinatorischen Schaltkreis darzustellen, geht man ähnlich induktiv vor wie beim Aufbau der Formeln:

- Eine atomare Formel x_i wird durch einen einzelnen Eingangsknoten X_i beschrieben
- Die Negation einer Teilformel wird erzeugt, indem eine Kante von dem die Teilformel repräsentierenden Teilschaltkreis in ein NOT-Gatter geführt wird.
- Die Konjunktion (Disjunktion) zweier Teilformeln wird erzeugt, indem von den die Teilformeln repräsentierenden Teilschaltkreisen jeweils eine Kante in ein AND-Gatter (OR-Gatter) geführt wird.
- Repräsentiert ein Teilschaltkreis bereits die gesamte Formel, so wird noch eine Kante zum Ausgangsknoten Y_i eingefügt.

Schaltkreise von Teilformeln, die in mehreren der zu repräsentierenden Formeln auftreten, müssen nicht mehrmals erzeugt werden. Es genügt, eine zweite ausgehende Kante einzufügen. Abbildung 2 zeigt einen kombinatorischen Schaltkreis, der die aussagenlogischen Formeln $\neg(x_1 \wedge x_2)$ und $(x_1 \wedge x_2) \vee x_3$ repräsentiert.

Sequentielle Schaltkreise

Viele wichtige Berechnungen, wie z.B. das Addieren und Multiplizieren von ganzen Zahlen[3], lassen sich mithilfe von kombinatorischen Schaltkreisen

[3] siehe [5, Kapitel 9]

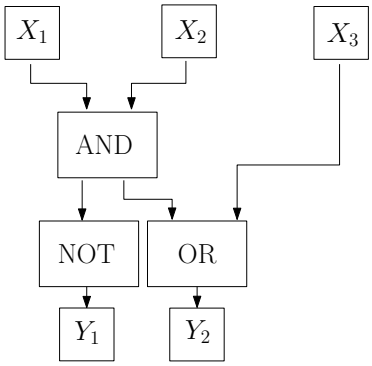

Abbildung 2: Kombinatorischer Schaltkreis für eine Funktion aus $\{0,1\}^3 \to \{0,1\}^2$

durchführen. Für viele komplexere Anwendungen ist es jedoch notwendig, Berechnungen anzustellen, die mehrere Schritte umfassen. Dies kann beispielsweise mithilfe von *sequentiellen Schaltkreisen* bewerkstelligt werden. Dabei werden zusätzlich zu den bereits bekannten Schaltkreiselementen *speichernde Elemente* eingefügt. Diese fungieren gleichzeitig als Ein- und Ausgang des Schaltkreises und können jeweils einen Wahrheitswert (0 oder 1) speichern. Somit ermöglichen sie es, dass Teile des Ergebnisses einer Berechnung Grundlage für die nächste Berechnung sein können. Das Ergebnis einer Berechnung hängt also nicht nur von der aktuellen Belegung an den Schaltkreiseingängen ab, sondern auch vom aktuellen Inhalt der speichernden Elemente. Man spricht dabei auch vom *Zustand* des Schaltkreises. Das Verhalten eines solchen sequentiellen Schaltkreises wird beschrieben durch eine boolsche Funktion für jeden Ausgang und jedes speichernde Element. Außerdem muss der Anfangszustand bekannt sein, damit ein eindeutiges Verhalten bei jeder Ausführung gewährleistet wird.
Im nächsten Abschnitt sollen Modelle eingeführt werden, mit denen Model Checking-Algorithmen arbeiten können. Außerdem wird erklärt, wie diese aus den tatsächlichen Schaltkreisen erzeugt werden können.

2.4 Modelle

In diesem Abschnitt werden Modelle vorgestellt, die das Verhalten der unterliegenden Systeme möglichst akkurat und mathematisch eindeutig beschreiben sollen. Für sequentielle Schaltkreise beispielsweise bedeutet dies, dass die Logik des Schaltkreises exakt abgebildet wird, während elektrische Eigenschaften wie die Gatterverzögerungszeiten vernachlässigt werden. Es existieren komplexere Modelle, die auch das physikalische Verhalten abbilden, auf diese

wird an dieser Stelle jedoch nicht näher eingegangen. Im Normalfall ist es jedoch erstrebenswert, das zu verifizierende System möglichst exakt zu beschreiben, um die beim Modelchecking erzielten Ergebnisse direkt auf das zugrunde liegende System übertragen zu können. So kann beispielsweise ein im Modell gefundenes Gegenbeispiel Hinweise auf den Fehlerort im System bieten, und ein fehlerfreies Modell repräsentiert ein fehlerfreies System. Für diese Zwecke stehen verschiedene Typen von Modellen zur Verfügung. Viele davon basieren auf den sog. *Transitionssystemen*.

Transitionssysteme

Diese können auf unterschiedliche Art und Weise definiert werden.
Im Rahmen dieser Arbeit dient folgende Definition als Ausgangspunkt. Es handelt sich dabei um eine leichte Vereinfachung der in [2] angegebenen Definition. An dieser Stelle wird auf die dort eingeführten *Aktionslabels* verzichtet, da diese im Rahmen dieser Arbeit keine weitere Rolle spielen.

Definition 7 (Transitionssystem TS)**.** Ein *Transitionssystem* TS ist ein Tupel $(S, \rightarrow, I, AP, L)$, und

- S ist eine Menge von Zuständen,
- $\rightarrow \subseteq S \times S$ ist eine Transitions- oder Übergangsrelation,
- $I \subseteq S$ ist eine Menge von Initialzuständen,
- AP ist eine Menge atomarer Eigenschaften, und
- $L : S \rightarrow 2^{AP}$ ist eine Labelingfunktion.*

* Implizit gilt für alle $a \in AP$, $s \in S : a \notin L(s) \rightarrow \neg a \in L(s)$

Sind S und AP außerdem endliche Mengen, so spricht man von einem endlichen Transitionssystem.
Die Zustände des Transitionssystems entsprechen den möglichen Zuständen des modellierten Systems. Ein Zustand in einem sequentiellen Schaltkreis wird z.B. wie gehabt über den Inhalt seiner speichernden Elemente und die aktuelle Eingangsbelegung beschrieben. Es ist auch möglich, Systeme auf einem abstrakteren Level zu modellieren, beispielsweise eine Ampel, deren Zustände für die aktuell leuchtenden Lampen stehen. Dieses Beispiel wird in Abbildung 3 veranschaulicht. Hier ist zu sehen, wie typischerweise die Zustände und die Übergangsrelation eines TS mit den Knoten und Kanten eines gerichteten Graphen assoziert werden. Die Labels eines Zustandes werden in Mengenschreibweise daneben gestellt.

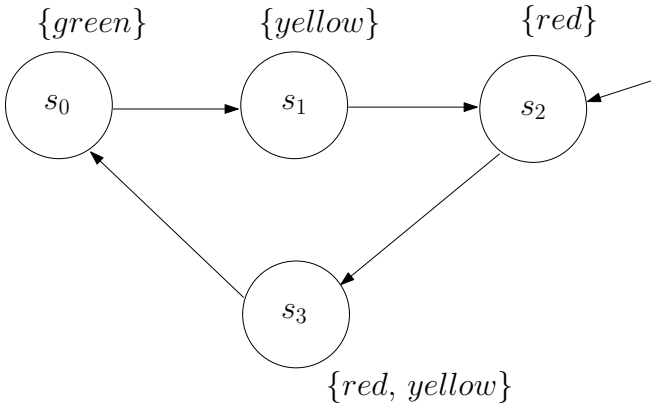

Abbildung 3: TS, das vereinfacht das Verhalten einer Verkehrsampel modeliert. Ausführung beginnt in s_2 (rote Ampelphase).

Die Initialzustände des TS - im Graphen mit einer eingehenden Kante ohne Ausgangsknoten gekennzeichnet - repräsentieren diejenigen Zustände, die das System zu Beginn der Ausführung annehmen kann. Jedesmal, wenn sich der Zustand des Systems ändert - z.B. durch Einwirkung von außen oder mit jedem neuen Berechnungsschritt - wird auch das TS entlang der Übergangsrelation in einen neuen Zustand überführt. Um sog. *Deadlocks* zu vermeiden - Situationen, in denen das System keinen weiteren Schritt absolvieren kann - muss jeder Zustand mindestens eine ausgehende Kante haben. Ein Pfad in einem TS ist eine Folge $s_1, s_2, \ldots, s_n$ mit $s_i \in S$ für alle i, bei der für alle $1 \leq i \leq n-1$ gilt, dass $s_i \rightarrow s_{i+1}$. Jeder Pfad beschreibt eine mögliche Ausführungsfolge des TS.
Je nach Anwendungsgebiet können Transitionssysteme um unterschiedliche Eigenschaften erweitert werden, etwa um probabilistische Elemente oder Echtzeiteigenschaften. An dieser Stelle soll jedoch nicht weiter auf diese Erweiterungen eingegangen werden.
Der nächste Abschnitt beschäftigt sich mit Eigenschaften, deren Gültigkeit in einem System mithilfe von Model Checking überprüft werden soll.

2.5 Eigenschaften

Wie bereits in Abschnitt 2.1 erörtert wurde, wird beim Model Checking die Gültigkeit bestimmter *Eigenschaften* in einem System überprüft. Eigenschaften beschreiben das gewünschte Verhalten des betrachteten Systems und werden als Formeln über der Menge AP der atomaren Eigenschaften formuliert. Für jeden Zustand s_i wird dabei $L(s_i)$ als Belegung über AP interpretiert: $\alpha_{s_i}(x) = 1$ gdw. $x \in L(s_i)$. Gilt $\alpha_{s_i} \models \Phi$, so sagt man s_i *erfüllt*

Φ und schreibt kurz $s_i \models \Phi$.
Zur Formulierung von Eigenschaften stehen verschiedene *Temporallogiken* zur Verfügung. Dabei wird die Aussagenlogik um temporale Operatoren erweitert, die beispielsweise ausdrücken, dass eine Formeln über *AP immer* (in jedem Zustand eines Pfades) oder *irgendwann* (in mindestens einem Zustand eines Pfades) gelten soll. Die Syntax und Semantik dieser Temporallogiken wird in [2, Kapitel 4/5] erläutert.
Üblicherweise werden drei Klassen von Eigenschaften unterschieden: *Sicherheits*-Eigenschaften und *Invarianten*, *Liveness*-Eigenschaften und *Fairness*-Eigenschaften. Während im späteren Verlauf dieser Arbeit nur den Invarianten Beachtung geschenkt wird, sollen an dieser Stelle alle drei Klassen von Eigenschaften kurz vorgestellt werden. Eine ausführlichere Übersicht über die unterschiedlichen Eigenschaften kann z.B. in [2, Kapitel 3] gefunden werden.

Sicherheits-Eigenschaften

Sicherheits-Eigenschaften sind, allgemein gesagt, Eigenschaften, die fordern, dass das System niemals „etwas Schlechtes tut“. Welches Verhalten dabei als *schlecht* gilt, hängt stark vom betrachteten System ab. Für die Ampel aus Abbildung 3 wäre ein schlechtes (oder *fehlerhaftes*) Verhalten beispielsweise, dass eine Rotphase unter Auslassung der Gelbphase direkt auf eine Grünphase folgt. Für gewöhnlich gibt es in jedem System eine ganze Reihe dieser unerwünschten Verhaltensmuster, die vermieden werden sollen.
Sicherheits-Eigenschaften werden oft als *Invarianten* formuliert. Invarianten sind Eigenschaften, die fordern, dass bestimmte Bedingungen in jedem erreichbaren Zustand des Systems gelten müssen. Damit eine Eigenschaft Φ in einem System invariant gültig ist, muss sie zunächst in allen Initialzuständen des Systems gelten. Weiterhin muss für jede Transition $s_i \rightarrow s_j$ gelten: $s_i \models \Phi$ gdw. $s_j \models \Phi$. Es gibt also keine gültige Ausführungsfolge des Systems, die in einem Initialzustand beginnt und bei der in irgendeinem Zustand $\neg\Phi$ gilt. Erfüllt ein System eine invariante Eigenschaft nicht, so exisitiert folglich immer ein endlich langes *Gegenbeispiel* - ein möglicher Pfad im System, in dessen i-ten Zustand $\neg\Phi$ gilt für ein $i \in \mathbb{N}$.

Liveness-Eigenschaften

Sicherheits-Eigenschaften fordern, dass das System niemals etwas *Schlechtes* tut. Diese Anforderung können die meisten Systeme leicht erfüllen, indem sie *gar nichts* tun. Offensichtlich ist dies aber nicht das gewünschte Verhalten der meisten Systeme. Deshalb gibt es die Liveness-Eigenschaften, die fordern, dass das System während seiner Ausführung einen gewissen Fortschritt

macht. Für das Ampelbeispiel wäre also eine Liveness-Eigenschaft, dass die Ampel irgendwann grün wird. Gegenbeispiele für Liveness-Eigenschaften sind unendliche Pfade, auf denen der gewünschte ‚gute'Zustand niemals eintritt.

Fairness-Eigenschaften

Eine weitere wichtige Anforderung an das Verhalten von Systemen ist die Fairness. Diese ist vorallem dann von Bedeutung, wenn ein System mehr als einen ‚guten' Zustand hat. Eine Fairness-Eigenschaft fordert, dass alle diese ‚guten' Zustände im Laufe einer Ausführungsfolge des Systems unendlich oft erreicht werden können. Ein Beispiel für Abbildung 3 wäre es also zu fordern, dass sowohl unendlich viele Rot- als auch unendlich viele Grünphasen auftreten. Ein Gegenbeispiel für eine Fairness-Eigenschaft wäre ein unendlich langer Pfad, ab dessen i-ten Zustand (für ein $i \in \mathbb{N}$) nur noch eine Teilmenge der guten Zustände erreicht wird.

Gemeinsam können diese drei Klassen von Eigenschaften das gewünschte Verhalten eines Systems festlegen. Es ist unerlässlich, bei der Ausarbeitung der zu verifizierenden Eigenschaften mit großer Sorgfalt und Umsicht zu Werke zu gehen. Nicht umsonst gilt die korrekte Formulierung der Spezifikation als einer der heikelsten Aufgabenstellungen im Verifikationsprozess, und falsche oder ungenaue Anforderungen sind oft genau so sicherheitskritisch wie Fehler im System selbst.
Im nächsten Abschnitt werden das sog. *Erfüllbarkeitsproblem der Aussagenlogik* sowie Algorithmen zu dessen Lösung vorgestellt. Es wird sich zeigen, dass viele Model Checking-Probleme auf die Lösung des Erfüllbarkeitsproblems zurückgeführt werden können.

2.6 Das Erfüllbarkeitsproblem der Aussagenlogik

Beim Erfüllbarkeitsproblem der Aussagenlogik (kurz *SAT-Problem* handelt es sich um die Frage, ob eine bestimmte aussagenlogische Formel erfüllbar ist, das heißt ob eine Belegung α existiert, so dass die gesamte Formel zu *true* bzw. 1 evaluiert.
Da jede gegebene Formel nur endlich viele Variablen enthält, existieren auch nur endlich viele mögliche Belegungen dieser Variablen. Ein naiver Ansatz zur Lösung des SAT-Problems wäre es also, alle möglichen Belegungen durchzuprobieren. Der Algorithmus terminiert, wenn entweder eine erfüllende Belegung gefunden wurde, oder alle Belegungen getestet wurden und keine davon die Formel erfüllt hat. Allerdings existieren bei n Variablen 2^n mögliche Belegungen, die im schlimmsten Fall alle ausprobiert werden müssen. Die Laufzeit des

Algorithmus ist also exponentiell[4] in der Anzahl der Variablen. Leider ist das SAT-Problem außerdem *NP-vollständig*[5], was bedeutet, dass (soweit bekannt) jeder Algorithmus im schlechtesten Fall exponentielle Zeit benötigt, um es zu lösen. Jedoch existieren Algorithmen, die die mittlere Laufzeit deutlich senken können.
Algorithmen, die in der Lage sind, das SAT-Problem zu lösen, werden auch als *SAT-Solver* bezeichnet. Die in der Praxis eingesetzten modernen SAT-Solver basieren auf dem 1962 von Martin Davis, Hilary Putnam, George Logemann und Donald W. Loveland vorgestellten *DPLL-Algorithmus* [7]. Die genaue Funktionsweise dieser Algorithmen ist für das Verständnis dieser Arbeit nicht relevant. Es sollen an dieser Stelle nur einige Mechanismen kurz umrissen werden, die in Kapitel 3 wieder zur Sprache kommen.
SAT-Solver akzeptieren in der Regel eine aussagenlogische Formeln in KNF (vergleiche auch Abschnitt 2.2) als Eingabe. Anschließend wird nach einer Belegung gesucht, die diese Formel erfüllt. Dabei wird nacheinander jeder einzelnen Variable eine Belegung zugewiesen, es wird also anfangs stets nur eine Belegung für eine Teilmenge der Variablen betrachtet.
Zunächst wird versucht, möglichst viele Variablenbelegungen eindeutig herzuleiten – zu *propagieren*. Wenn etwa eine Klausel von der aktuellen Belegung noch nicht erfüllt wird und nur noch ein (unbelegtes) Literal enthält, so gibt es nur eine Möglichkeit, dieses Literal zu belegen, wenn die gesamte Formel noch erfüllt werden soll. Sind keine solchen eindeutigen Schritte mehr möglich, so werden *Entscheidungen* getroffen, indem nach bestimmten Heuristiken eine Variable ausgewählt und – wieder einer Heuristik folgend - mit 0 oder 1 belegt wird. Als Erweiterung des DPLL-Algorithmus sind die meisten Algorithmen außerdem in der Lage, dazu zu ‚lernen', wenn die bei der Suche nach einer erfüllenden Belegung getroffenen Entscheidungen zu Widersprüche führen. Das so erhaltene Wissen – die sog. *Konfliktklausel* – kann anschließend verwendet werden, um im weiteren Verlauf der Suche bessere Entscheidungen treffen zu können. Einen Überblick über aktuelle SAT-Solving-Techniken liefert z.B. [8].
Im nächsten Abschnitt wird mit dem *Bounded Model Checking* eine Technik vorgestellt, die SAT-Solver zur Verifikation von Eigenschaften in Transitionssystemen einsetzt.

[4]Steigt die Anzahl der Variablen linear, so erhöht sich die Laufzeit des Algorithmus exponentiell.
[5]Beweis und Erklärung siehe [6].

2.7 Bounded Model Checking

Beim *Bounded Model Checking* (kurz *BMC*) handelt es sich um eine Technik, die für ein gegebenes Modell M eines Systems und eine Eigenschaft P schrittweise nach Gegenbeispielen der Länge $k \in \mathbb{N}$ sucht, wobei k in jedem Schritt um eins erhöht wird. Während unterschiedliche Implementierungen von BMC-Prozeduren existieren, wird in der Praxis doch meist der auch in [9] vorgestellte Ansatz auf Grundlage von SAT-Solvern verwendet, da er sich als sehr effizient erwiesen hat.
Dabei wird eine aussagenlogische Formel erstellt, die das Modell M beschreibt und genau dann erfüllbar ist, wenn ein Pfad der Länge k exisitert, der die Eigenschaft P verletzt. Diese Formel setzt sich im Wesentlichen aus drei Teilformeln zusammen:

- $I(s_i)$ beschreibt die Initialzustände des Modells und ist genau dann erfüllt, wenn es sich bei s_i um einen solchen handelt.
- $T(s_i, s_{i+1})$ beschreibt die Transitionsrelation des Modells und ist genau dann erfüllt, wenn zwischen s_i und s_{i+1} ein gültiger Übergang exisitert.
- $P(s_i)$ beschreibt die zu verifizierende invariante Eigenschaft und ist genau dann erfüllt, wenn diese Eigenschaft in s_i gültig ist.

Hierbei ist s_i der i-te Zustand eines Pfads. Dabei ist zu beachten, dass das Modell hier *binär* kodiert wird. Jeder Zustand wird repräsentiert durch einen eindeutigen Vektor boolscher Variablen, der die in diesem Zustand gültigen atomaren Eigenschaften beschreibt. Eine aussagenlogische Formel, die einen Pfad der Länge k beschreibt, der in einem Initialzustand beginnt und dessen letzter Zustand P verletzt, sieht also folgendermaßen aus:

$$\textbf{BMC'}_k \quad = \quad I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

Da zunächst $\textbf{BMC'}_0$ überprüft und k anschließend in jedem Schritt um eins erhöht wird, findet man dabei immer das kürzeste existierende Gegenbeispiel bezüglich einer Eigenschaft (oder gar keines). Wenn also $\textbf{BMC'}_k$ überprüft wird, ist bereits bekannt, dass kein Gegenbeispiel der Länge $i < k$ existiert, da dieses bereits im i-ten Schritt gefunden worden wäre. Es kann deshalb angenommen werden, das P in allen Zuständen ausser dem letzten gültig ist. Dadurch erhält man die folgende Formel:

$$\textbf{BMC}_k \quad = \quad I(s_0) \wedge P(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

Die Einschränkung auf Pfade, bei denen lediglich der letzte Zustand P verletzt, beschleunigt die Suche nach einer Lösung, da so deutlich weniger mögliche

Pfade berücksichtigt werden müssen.
BMC ist ein *korrektes* Verfahren zur Suche nach Gegenbeispielen in Modellen. *Korrektheit* bedeutet hier, dass, wenn der Algorithmus terminiert, ein korrektes Ergebnis ausgegeben wird. Allerdings terminiert der BMC-Algorithmus nur dann, wenn ein Gegenbeispiel entdeckt wurde. Da in fehlerfreien Modellen keine solchen Gegenbeispiele exisitieren, ist das Verfahren also im Allgemeinen nicht *vollständig* - es terminiert nicht für jede Eingabe. Für endliche Transitionssysteme kann BMC vollständig gemacht werden, indem man zusätzlich fordert, dass alle Zustände eines Pfades verschieden sein müssen. Jeder Pfad – und somit auch jedes Gegenbeispiel – kann dann maximal so viele Schritte enthalten, wie das betrachtete System Zustände besitzt. Wenn k die Anzahl der Zustände in S erreicht – den sog. *Durchmesser* des Systems – ohne ein Gegenbeispiel zu finden, so kann man sicher sein, dass in dem System keine (erreichbaren) Zustände existieren, die P verletzen. Allerdings ist der Durchmesser vieler Systeme sehr groß, und die SAT-Prozeduren für entsprechend lange Formeln haben eine sehr hohe Laufzeit, weshalb dieses Verfahren in der Regel nicht angewendet wird. Im Rahmen dieser Arbeit wurde mit der *k-Induktion* ein effizienteres Verfahren implementiert, um Vollständigkeit für BMC-Prozeduren zu erreichen. Bevor dieses Verfahren in Kapitel 4 erläutert wird, soll im Folgenden der verwendete SAT-Solver *iSAT* vorgestellt werden.

3 iSAT

iSAT ist ein SMT-Solver, der im Rahmen des AVACS-Projektes für automatische Verifikation und Analyse komplexer Systeme [10] unter Anderem am Lehrstuhl für Rechnerarchitektur der Universität Freiburg entwickelt wurde. Der in [11] vorgestellte Solver kann sowohl einzelne Formeln als auch BMC-Probleme lösen. Bei *SMT-Solvern* handelt es sich um eine Erweiterung der in Abschnitt 2.6 vorgestellten SAT-Solver. Bevor im Folgenden auf die Besonderheiten von iSAT eingegangen wird, sollen zunächst kurz das SMT-Problem sowie die grundlegende Funktionsweise eines SMT-Solvers vorgestellt werden.

3.1 SAT modulo Theory

Die bisher behandelten SAT-Solver sind nur in der Lage, rein aussagenlogische Formeln zu lösen. In realen Anwendungen reicht dies jedoch oft nicht aus, um das vorliegende System akkurat zu beschreiben. Das *SMT-Problem* (*SAT modulo theory Problem*) ist eine Erweiterung des Erfüllbarkeitsproblems um sog. *Theorieatome*. Theorieatome sind Gleichungen und Ungleichungen arithmetischer Ausdrücke, wie z.B. $(a + b = c)$ oder $(x > y)$. Die so konstruierten

Formeln können also auch Aussagen über ganze oder sogar reele Zahlen treffen, wobei hier zunächst nur Theorieatome mit ausschließlich linearen Operationen betrachtet werden sollen.
Auch zur Lösung des SMT-Problems existieren diverse Algorithmen, die *SMT-Solver* genannt werden. Einen einfacher Ansatz zur Lösung von SMT-Formeln stellt das *Lazy Theorem Proving* dar. Dabei wird zunächst eine boolsche Abstraktion der zu prüfenden Formel erzeugt, indem jedes Theorieatom durch eine boolsche Hilfsvariable ersetzt wird. Mithilfe eines herkömmlichen SAT-Solvers wird dann die Erfüllbarkeit dieser abstrahierten Formel überprüft. Ist bereits die Abstraktion unerfüllbar, so gilt dies zwangsläufig auch für die zugrundeliegende Formel. Wird für die Abstraktion eine erfüllende Belegung gefunden, so wird diese an einen sog. *Theory-Solver* weitergereicht. Dieser ist in der Lage zu überprüfen, ob diese *Widersprüche* enthält (wie z.B. $(x > y)$ und $(y > x)$, was nicht gleichzeitig wahr sein kann). Falls die gefundene Belegung widerspruchsfrei ist, so ist die ursprüngliche Formel erfüllbar. Falls nicht, so wird die Abstraktion um eine Klausel ergänzt, die Informationen darüber enthält, welche Theorieatome den Widerspruch verursacht haben. Anschließend wird erneut mithilfe des SAT-Solvers nach einer erfüllenden Belegung gesucht. Dieser Vorgang wird solange wiederholt, bis ein eindeutiges Ergebnis gefunden wurde. Eine ausführlichere Erklärung und Beispiele zum Thema Lazy Theorem Proving liefert z.B. [12, Kapitel 17].
Die Theorieatome können auch um einige *nicht-lineare* Operationen wie die Multiplikation von Variablen (z.B. $a \cdot b = c$) erweitert werden. In vielen realen Systemen werden allerdings auch *transzendente* Funktionen zur korrekten Modellierung benötigt. Viele physikalische Zusammenhänge etwa lassen sich nur mithilfe der *Sinus* und *Cosinus* Funktion darstellen, und auch die Exponentialfunktion wird zur Darstellung vieler Sachverhalte benötigt. Sobald eine Formel allerdings solche Funktionen enthält, kann ein Theorie-Solver nicht mehr in allen Fällen eindeutig feststellen, ob eine gefundene Belegung widerspruchsfrei ist oder nicht – das Problem wird *unentscheidbar*. Um auch in dieser Domäne des Unentscheidbaren möglichst gute Ergebnisse erzielen zu können, implementiert iSAT einen anderen Ansatz zur Lösung des SMT-Problems – die *Intervall-Arithmetik*. Bevor diese jedoch in Abschnitt 3.3 vorgestellt wird, soll im nächsten Abschnitt zunächst die von iSAT verwendete *Syntax* erklärt werden.

3.2 iSAT Syntax

iSAT akzeptiert Formeln in einem speziellen Eingabeformat, dem *.hys*-Format. In diesem Format muss jede Variable vor ihrer Verwendung *deklariert* werden. Bei der Deklarierung erhält jede Variable einen eindeutigen *Namen*, einen *Typ*

(boolsch, ganzzahlig oder reellwertig) und einen *Wertebereich*. Bei boolschen Variablen umfasst der Wertebereich genau die Menge $\{0, 1\}$, für ganzzahlige (reelle) Zahlen ein beliebiges *Intervall* ganzzahliger (reeller) Werte.
Da iSAT sowohl einzelne Formeln als auch BMC-Probleme lösen kann, können auch die akzeptierten Eingabedateien auf zwei verschiedene Arten definiert werden. Soll eine einzelne Formel gelöst werden, so enthält die Datei neben der Deklaration der Variablen nur diese Formel, eine sog. *Expression*. Das aus [13] entnommene Beispiel 1 zeigt eine solche Datei, die den Satz von Pytagoras (also die Formel $a \cdot a + b \cdot b = c \cdot c$) beschreibt. a, b und c sind dabei jeweils als ganzzahlige Werte aus dem Intervall $[0, 100]$ definiert.

Beispiel 1 (.hys-Datei für eine einzelne Formel)**.**

```
DECL
      int [0, 100] a, b, c;

EXPR
      a*a + b*b = c*c;
```

Soll stattdessen ein BMC-Problem formuliert werden, so besteht die Datei neben der Deklaration aus drei weiteren Abschnitten, die jeweils eine SMT-Formel enthalten. Der Abschnitt `INIT` beschreibt die Anfangszustände des betrachteten Modells. `TRANS` stellt die Übergangsrelation dar, wobei für jede Variable x x' den Wert der Variable im nächsten Schritt beschreibt. `TARGET` schließlich beschreibt das ‚Ziel' des gesuchten Pfades und entspricht somit der negierten invarianten Eigenschaft $\neg P$ vom allgemeinen BMC-Ansatz (vergleiche Abschnitt 2.7). Beispiel 2 zeigt eine .hys-Datei, die einen einfachen Zähler beschreibt. Eine Variable x wird zunächst mit 0 initialisiert und anschließend in jedem Schritt um eins erhöht. Überprüft wird dabei, ob x irgendwann den Wert 50 erreicht. Die BMC-Prozedur würde also nach 50 Schritten eine Lösung finden.

Beispiel 2 (.hys-Datei für ein Transitionssystem)**.**

```
DECL
      int [0, 100] a;

INIT
      x = 0;
```

```
TRANS
      x' = x + 1;

TARGET
      x = 50;
```

Die in einer .hys-Datei formulierten SMT-Formeln können eine Großzahl von Operatoren in nahezu belieber Kombination enthalten – Details zu den von iSAT akzeptierten Eingabeformeln bietet [13]. Der Solver selbst allerdings kann – wie die meisten SAT-Solver – nur auf einer Formel in KNF operieren (vergleiche Abschnitt 2.2). Analog zur bereits erwähnten Tseitin-Transformation existieren Verfahren, die beliebige SMT-Formeln in eine solche KNF umwandeln können. Neben der aussagenlogischen Struktur der Formel werden dabei auch die Theorieatome selbst umgeformt. Auf diese Art und Weise wird die Eingabeformel der *internen Syntax* von iSAT angepasst, bevor der eigentliche Solver mit seiner Arbeit beginnt.

Definition 8 (Interne Syntax von iSAT)**.**

- Eine *Formel* ist eine Konjunktion beliebig vieler *Klauseln*.
- Eine *Klausel* ist eine Disjunktion beliebig vieler *Atome*.
- Ein *Atom* ist entweder eine *Begrenzung* oder eine *Gleichung*.
- Eine *Begrenzung* hat die Form *Variable R rationale Konstante*, mit $R \in \{<, \leq, =, \geq, >\}$.
- Eine *Variable* ist entweder eine *reele Variable* oder eine *boolsche Variable*.
- Eine *Gleichung* ist entweder ein *Triplet* oder ein *Paar*.
- Ein *Triplet* hat die Form *reele Variable* = *reele Variable B reele Variable*, wobei B ein binärer Operator ist ($+$, $-$, $\cdot$,...).
- Ein *Paar* hat die Form *reele Variable* = U *reele Variable*, wobei U ein unärer Operator ist (sin, cos,...)

Sobald alle Formeln entsprechend dieser Syntax umgeformt wurden, kann iSAT das entsprechende BMC-Problem mithilfe der *Intervall-Arithmetik* lösen.

3.3 Intervall-Arithmetik

Wie im letzten Abschnitt besprochen wird jeder in iSAT deklarierten Variable ein möglicher Wertebereich, ein sog. Intervall, zugeordnet. Mithilfe dieser Intervalle werden – analog zu den herkömmlichen SAT-Solvern – Werte propagiert und Entscheidungen getroffen. Bei der *Intervall Constraint Propagation* werden anhand der zu erfüllenden Klauseln die gegebenen Intervalle so weit wie möglich *eingeschränkt*. Es seien beispielsweise drei ganzzahlige Variablen $x \in [3, 5]$, $y \in [1, 3]$ und $z \in [3, 9]$ gegeben, und es soll außerdem gelten, dass $x + y = z$. Da $x + y$ minimal 4 und maximal 8 ergibt (Summe der beiden unteren bzw. oberen Intervallgrenzen), kann z die Werte 3 und 9 in dieser Konstellation niemals erreichen. Das Intervall von z kann also auf $[4, 8]$ eingeschränkt werden. Wenn keine weiteren Einschränkungen mehr propagiert werden können, muss eine Entscheidung getroffen werden. Anders als beim herkömmlichen SAT-Solving gibt es aber nicht mehr nur zwei Werte, die eine Variable annehmen kann, sondern eine ganze Reihe – im Falle von reellwertigen Variablen sogar unendlich viele. Anstatt die ausgewählte Variable direkt mit einem Wert zu belegen, wird deshalb das entsprechende Intervall geteilt. Auch dafür existieren wieder verschiedene Heuristiken, der Einfachheit halber soll an dieser Stelle jedoch angenommen werden, dass die Teilung in der Mitte des Intervalls erfolgt. Anschließend wird entschieden, ob zunächst die obere oder die untere Hälfte des Intervalls betrachtet werden soll. Diese Teilung des Intervalls wird später wiederholt, solange, bis kein Intervall mehr breiter als ein vorher festgelegter Grenzwert ist. Gelingt dies, ohne auf einen Widerspruch zu treffen, so wurde eine *Candidate Solution* (Kandidatenlösung) entdeckt. Das bedeutet, dass die gefundenen Intervalle eventuell eine Lösung beinhalten. Wird eine Candidate Solution gefunden, bedeutet dies jedoch nicht zwangsläufig, dass auch eine konkrete Lösung des Problems existiert. Beispiel 3 veranschaulicht diesen Sachverhalt.

Beispiel 3 (Candidate Solution ohne konkrete Lösung)**.**
Zu lösen sei die Formel $F = (x + y > z) \wedge (x + y < z)$. Grenzwert für die Aufteilung der Intervalle sei 2.
Offensichtlich ist F unerfüllbar. iSAT liefert die Kandidatenlösung $x \in [0, 1]$, $y \in [0, 1]$ und $z \in [0, 2]$.
Wählt man aus diesen Intervallen $x_1 = 1$, $y_1 = 1$ und $z_1 = 1$, so wird mit $(x_1 + y_1 > z_1)$ die erste Klausel von F erfüllt. Wählt man hingegen $x_2 = 0$, $y_2 = 1$ und $z_2 = 2$, so wird mit $(x_2 + y_2 < z_2)$ die zweite Klausel von F erfüllt. Die Lösungsintervalle enthalten also Belegungen, die alle Klauseln von F erfüllen, auch wenn keine einzelne Belegung existiert, die alle Klauseln erfüllt.

Da das Lösen von nicht-linearen Gleichungen wie bereits erwähnt unentscheid-

bar ist, wird in der Regel keine exakte Lösung gefunden. Beim Auftreten von Widersprüchen werden wie auch beim herkömmlichen SAT-Solving die am Widerspruch beteiligten Belegungen rückgängig gemacht, und das dadurch erlernte neue Wissen als sog. *Konfliktklausel* zur Formel hinzugefügt.
Ausführlichere Informationen zur Funktionsweise von iSAT können aus [11] entnommen werden.

3.4 Inkrementelles Lösen

Eine weitere Technik, die von iSAT, aber auch von anderen modernen BMC-Solvern implementiert wird, ist das *inkrementelle Lösen*. Dabei wird ausgenutzt, dass beim BMC nacheinander viele Formeln gelöst werden müssen, die in weiten Teilen identisch sind. Dadurch ist auch die Wahrscheinlichkeit sehr hoch, immer wieder auf die gleichen Widersprüche zu stoßen. Die beim Lösen einer Formel gelernten Konfliktklauseln können also auch das Lösen der nachfolgenden Formeln beschleunigen. Beim inkrementellen Lösen wird deshalb die Formel nicht in jedem Schritt neu definiert, sondern jeweils auf der vorherangehenden aufbauend formuliert.
Solange in jedem Schritt nur neue Klauseln zur KNF hinzugefügt werden, können problemlos alle Konfliktklauseln des vorherigen Schritts übernommen werden: Da nach wie vor alle Klauseln der ursprünglichen Formel erfüllt werden müssen, können alle dort entdeckten Widersprüche erneut auftreten. Schwierig wird es erst, wenn Klauseln wieder entfernt werden – was beim BMC nötig ist, um die Teilformeln der Form $\neg P(s_k)$ im $k+1$-sten Schritt durch Teilformeln der Form $P(s_k) \wedge \neg P(s_{k+1})$ zu ersetzen. Konfliktklauseln, die bei der Betrachtung der nun entfernten Klauseln gefunden wurden, dürfen nicht in die neue Formeln übernommen werden, da nicht mehr garantiert ist, dass diese Widersprüche erneut auftreten können.
Es existieren verschiedene Ansätze, um dieses Problem zu umgehen. In iSAT werden zu diesem Zweck alle Teilformeln, die später wieder entfernt werden, markiert. So kann bereits beim Lernen der Konfliktklauseln entschieden werden, welche für den nächsten Schritt behalten werden und welche nicht. Für Teilformeln vom Typ $I(s_i)$ und $T(s_i, s_{i_1})$, die in jedem Schritt erneut enthalten sind, kann somit von den bereits erlernten Konfliktklauseln profitiert werden.
Die hier vorgestellte Implementierung der im nächsten Kapitel vorgestellten *k-Induktion* nutzt das inkrementelle Lösen analog zu dem in [1] diskutierten Ansatz aus.

4 k-Induktion

Wie bereits in Abschnitt 2.7 dargestellt, ist Bounded Model Checking eine effiziente Methode zum Finden von Gegenbeispielen. Die Korrektheit eines fehlerfreien Models (bezüglich einer bestimmten Eigenschaft P) kann jedoch nur mit sehr hohem Rechenaufwand nachgewiesen werden. In [1] stellen die Autoren mit der *k-Induktion* eine Methode vor, um dieses Problem zu umgehen. Diese Methode soll an dieser Stelle ausgehend von einem sehr einfachen, aber unvollständigen Ansatz schrittweise entwickelt werden.

4.1 Einfacher Ansatz

Ähnlich der vollständigen Induktion, bei der Beweise über den natürlichen Zahlen geführt werden, nutzt die k-Induktion einen Induktionsanfang (*Base*) und einen Induktionsschritt (*Step*), um eine Eigenschaft (die Induktionsbehauptung) eines Modells zu beweisen. Im Induktionsanfang wird zunächst gezeigt, dass die Eigenschaft in allen Initialzuständen des Modells gilt, es also keinen Zustand s_0 gibt, in dem $I(s_0) \wedge \neg P(s_0)$ gilt. Anschließend wird im Induktionsschritt bewiesen, dass, wenn in einem Zustand die Eigenschaft P gilt, sie auch im nächsten gelten wird. In Formeln ausgedrückt muss also für alle Zustände s_1 und s_2 gelten, dass $(P(s_1) \wedge T(s_1, s_2)) \Rightarrow P(s_2)$, was nach den in Abschnitt 2.2 eingeführten Notationen $P(s_2) \vee \neg((P(s_1) \wedge T(s_1, s_2))$ entspricht. Die Negation $\neg(P(s_2) \vee \neg((P(s_1) \wedge T(s_1, s_2))) \equiv P(s_1) \wedge T(s_1, s_2) \wedge \neg P(s_2)$ muss also unerfüllbar sein.
Um allgemein zu zeigen, dass diese Voraussetzungen auf allen Pfaden eines Systems erfüllt sind, muss die Unerfüllbarkeit der folgenden Formeln nachgewiesen werden:

k-Induktion Ansatz 1.

$$\begin{aligned} \textbf{Base} &:= I(s_0) \wedge \neg P(s_0) \\ \textbf{Step} &:= P(s_0) \wedge T(s_0, s_1) \wedge \neg P(s_1) \end{aligned}$$

Dies entspricht der Definition der in Abschnitt 2.5 eingeführten Invarianten. Die k-Induktion kann also beweisen, dass eine Eigenschaft P in einem System invariant gültig ist.

4.2 Erweiterter Ansatz

Häufig reicht Ansatz 1 jedoch nicht aus, um die Korrektheit eines Systems (bezüglich P) zu beweisen, nämlich dann, wenn es im System zwar einen

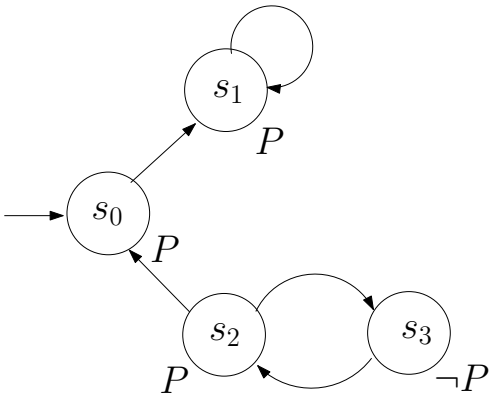

Abbildung 4: Der ‚schlechte' Zustand s_3 ist vom Initialzustand aus nicht erreichbar. Dennoch ist die Gleichung **Step** hier erfüllbar, da ein Übergang vom ‚guten' Zustand s_2 nach s_3 existiert.

Übergang von einem ‚guten' zu einem ‚schlechten' Zustand gibt, aber bereits dieser vorangehende ‚gute' Zustand von den Initialzuständen aus nicht erreichbar ist. Ein Beispiel für ein solches System zeigt Abbildung 4. Um dieses Problem zu umgehen, verlangt der erweiterte Ansatz, dass P für jeden Zustand eines Pfads der Länge k gilt (statt nur für einen einzelnen Zustand), bevor schließlich ein Zustand erreicht wird, in dem $\neg P$ gilt. Wie beim Bounded Model Checking wird dabei k in jedem Schritt erhöht. Die Formeln für den Induktionsanfang und -schritt müssen dazu folgendermaßen erweitert werden:

k-Induktion Ansatz 2.

$$\mathbf{Base}_k := I(s_0) \wedge \Big(\bigwedge_{i=0}^{k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \Big) \wedge \neg P(s_k)$$

$$\mathbf{Step}_k := \Big(\bigwedge_{i=0}^{k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \Big) \wedge \neg P(s_{k+1})$$

Ist für ein $k \in \mathbb{N}$ $\mathbf{Base}_k$ unerfüllbar, so existiert kein Pfad, der in einem Initalzustand beginnt und nach k Schritten zum ersten Mal einen Zustand erreicht, in dem $\neg P$ gilt. Jeder Pfad, der $\mathbf{Base}_k$ erfüllt, ist also ein Gegenbeispiel der Länge k. Wenn $\mathbf{Step}_k$ unerfüllbar ist für ein $k \in \mathbb{N}$, dann gibt es keinen Pfad der Länge k, auf dem in jedem Zustand P gilt und von dessen letztem Zustand aus ein Zustand erreicht werden kann, in dem $\neg P$ gilt.
Das Verfahren terminiert, wenn für eine Tiefe k $\mathbf{Base}_k$ erfüllbar ist (Das System ist nicht korrekt bezüglich P, da ein Gegenbeispiel gefunden wurde) oder $\mathbf{Base}_k$ und $\mathbf{Step}_k$ unerfüllbar sind (Das System ist korrekt bezüglich k).

Da die Korrektheit von Ansatz 2 nicht mehr intuitiv ersichtlich ist, soll sie im Folgenden bewiesen werden.

Beweis der Korrektheit

Es soll gezeigt werden, dass ein System korrekt ist bezüglich einer Eigenschaft P, wenn $\mathbf{Step}_k$ unerfüllbar ist für eine Tiefe $k \in \mathbb{N}$ und $\mathbf{Base}_i$ unerfüllbar ist für alle $i \leq k$. Ein System ist korrekt bezüglich P genau dann, wenn kein Pfad von einem Initalzustand zu einem Zustand exisitert, in dem $\neg P$ gilt.

Beweis durch vollständige Induktion:
Induktionsbehauptung: Sei $\mathbf{Step}_k$ unerfüllbar für ein $k \in \mathbb{N}$ und $\mathbf{Base}_i$ unerfüllbar für alle $i \leq k$. Dann existiert kein Pfad der Länge n von einem Initialzustand zu einem Zustand, in dem $\neg P$ gilt, für alle $n \in \mathbb{N}$. Das heißt, es existiert kein Gegenbeispiel der Länge n.

Induktionsanfang: $n \leq k$. Da $\mathbf{Base}_i$ unerfüllbar ist für $i \leq k$, ist bereits bewiesen, dass kein Gegenbeispiel der Länge k oder kürzer exisitiert.

Induktionsschritt: $n \geq k$, $n \to n+1$. Sei $s_0 s_1 \dots s_n$ ein Pfad der Länge n, der in einem Initialzustand beginnt. Laut Induktionsbehauptung gilt in jedem Zustand dieses Pfades P. Dann ist $s_{n-k} \dots s_n$ ein Pfad der Länge k, für den in jedem Zustand P erfüllt ist. Da $\mathbf{Step}_k$ unerfüllbar ist, gibt es keinen Pfad $s_0 \dots s_{k+1}$, so dass in s_{k+1} $\neg P$ gilt. Durch Variablen-Umbenennung kann man folgern, dass auch kein Pfad $s_{n-k} \dots s_n s_{n+1}$ existiert, so dass in s_{n+1} $\neg P$ gilt. Daraus folgt, dass es auch keinen Pfad $s_0 \dots s_{n+1}$ gibt, für den in s_{n+1} $\neg P$ gilt. Es existiert also kein Gegenbeispiel der Länge $n+1$.

q.e.d.

4.3 Uniqueness

Das in Abbildung 4 dargestellte Problem kann also mithilfe von Ansatz 2 umgangen werden. Betrachtet man allerdings Abbildung 5, so wird klar, dass auch dieser stärkere Ansatz noch immer nicht vollständig ist – durch Zyklen oder auch *Schleifen* im Transitionssystem kann $\mathbf{Step}_k$ für jede Tiefe k erfüllbar sein, obwohl vom Initialzustand aus kein Zustand erreicht werden kann, in dem $\neg P$ gilt.
Die Vollständigkeit des Verfahrens kann jedoch gewährleistet werden, wenn gefordert wird, dass alle Zustände des Pfades unterschiedlich sind. Diese

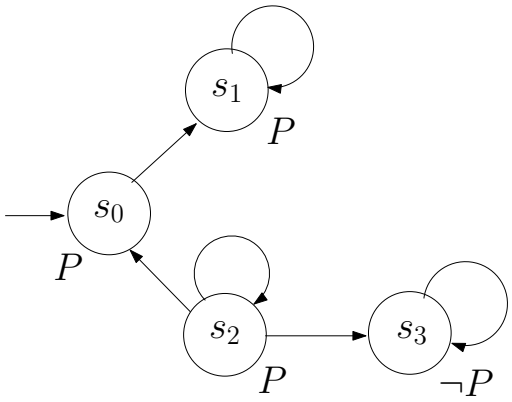

Abbildung 5: Der ‚schlechte' Zustand s_3 ist vom Initialzustand aus nicht erreichbar. Dennoch ist die Gleichung $\mathbf{Step}_k$ für kein $k \in \mathbb{N}$ unerfüllbar, da durch den ‚Selbstübergang' in s_2 beliebig lange Pfade entstehen, in deren letzten Zustand P verletzt wird.

Eigenschaft heißt *Uniqueness*. Die Formel, die diese Eigenschaft beschreibt, heißt $\mathbf{Unique}_k$ und fordert, dass sich die Vektoren der Zustandsvariablen für jedes Paar von Zuständen an mindestens einer Stelle unterscheiden. Dies geschieht mithilfe vom Teilformeln $U(s_i, s_j)$. $U(s_i, s_j)$ ist erfüllt genau dann wenn $s_i \neq s_j$.

k-Induktion Ansatz 3.

$$\begin{aligned}
\mathbf{Base}_k &:= I(s_0) \wedge \Big(\bigwedge_{i=0}^{k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \Big) \wedge \neg P(s_k) \\
\mathbf{Step}_k &:= \Big(\bigwedge_{i=0}^{k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \Big) \wedge \neg P(s_{k+1}) \\
\mathbf{Unique}_k &:= \bigwedge_{i \leq j \leq k} (s_i \neq s_{j+1}) = \bigwedge_{i \leq j \leq k} U(s_i, s_j)
\end{aligned}$$

Da hier nur Modelle mit endlich vielen Zuständen betrachtet werden, muss $\mathbf{Step}_k$ für ein Modell, dass die invariante Eigenschaft erfüllt, also spätestens dann unerfüllbar werden, wenn k der Gesamtanzahl der Zustände des Modells entspricht. Das Verfahren der k-Induktion ist somit vollständig. Algorithmus 1 veranschaulicht einen entsprechenden Algorithmus. Für jedes k wird in Zeile 2 zunächst überprüft, ob $\mathbf{Base}_k$ erfüllbar ist. Falls ja, so wurde ein Gegenbeispiel der Länge k gefunden, und das System ist bezüglich P *unsicher*. Ansonsten wird in Zeile 5 überprüft, ob die Konjunktion von $\mathbf{Step}_k$ und $\mathbf{Unique}_k$ unerfüllbar ist. Ist dies der Fall, so ist P in allen erreichbaren Zuständen des

Algorithm 1 k-Induktion

```
1: for k ∈ 0 ... ∞ do
2:   if satisfiable(Base_k) then
3:     return UNSAFE
4:   else
5:     if ¬satisfiable(Step_k ∧ Unique_k) then
6:       return SAFE
7:     end if
8:   end if
9: end for
```

Systems gültig, und das System ist *sicher* bezüglich P. Ist $\mathbf{Step}_k \wedge \mathbf{Unique}_k$ hingegen erfüllbar, so wird k um eins erhöht und die beiden Tests in Zeile 2 und 5 werden erneut durchgeführt, solange, bis der Algorithmus terminiert. An dieser Stelle soll noch angemerkt werden, dass das Hinzufügen von $\mathbf{Unique}_k$ die Anzahl der pro Solver-Lauf zu betrachtenden Klauseln und somit die Laufzeit stark erhöht. Um dem entgegenzuwirken, können zwei Verfeinerungen der Technik angewandt werden, die *dynamische Uniqueness* und die *starke Uniqueness*.

Dynamische Uniqueness

Bei der dynamischen Uniqueness werden nur diejenigen Klauseln aus $\mathbf{Unique}_k$ zur KNF hinzugefügt, die auch wirklich benötigt werden, also nur für diejenigen Zustände, die durch eine Schleife erreicht werden können. Um festzustellen, welche Klauseln aus $\mathbf{Unique}_k$ hinzugefügt werden müssen, wird zunächst die Erfüllbarkeit von $\mathbf{Step}_k$ geprüft. Wird dabei eine Lösung gefunden, so wird überprüft, ob zwei Zustände des Gegenbeispiels übereinstimmen. Sollte dies für zwei Zustände s_1 und s_2 der Fall sein, so wird $U(s_1, s_2)$ zur KNF hinzugefügt. Dies wird für alle Paare von identischen Zuständen wiederholt. Anschließend wird erneut $\mathbf{Step}_k$ (inklusive der neuen Klauseln) überprüft. Erst wenn eine Lösung gefunden wurde, die nur paarweise verschiedene Zustände enthält, wird k erhöht und mit dem nächsten Schritt der k-Induktion fortgefahren.

Starke Uniqueness

Eine andere Möglichkeit, die Anzahl der für die Uniqueness benötigten Klauseln zu reduzieren, ist es, die betrachteten Zustandsvariablen einzuschränken. Wie in 2.7 beschrieben, wird beim BMC jeder Zustand durch einen Vektor kodiert, der die Belegung aller atomaren Eigenschaften in diesem Zustand

beschreibt. Falls das betrachtete TS einen Schaltkreis beschreibt, so enthält dieser Vektor den Inhalt aller speichernden Elemente sowie die Eingangsbelegung des Schaltkreises. Da jedoch nur die speichernden Elemente den Zustand des Schaltkreises beschreiben (die Eingangsbelegung kann in jedem Schritt beliebig neu gewählt werden), ist es für die Uniqueness ausreichend, nur diese zu betrachten. Auf diese Art und Weise wird die Anzahl der Klauseln, die nötig ist, um $U(s_i, s_j)$ zu beschreiben, deutlich reduziert. Der Beweis, dass die Uniqueness auch mit dieser Beschränkung auf die sog. *Latch-Variablen* noch korrekt funktioniert, kann aus [1, Kapitel 4.2] entnommen werden.

In der in Kapitel 5 vorgestellten Implementierung der k-Induktion wurden sowohl die dynamische als auch die starke Uniqueness berücksichtigt.

4.4 k-Induktion für arithmetische Transitionssysteme

Bisher wurde beschrieben, wie die k-Induktion für rein boolsch kodierte Transitionssysteme funktioniert. Wie in Kapitel 3 beschrieben, enthält iSAT jedoch auch lineare und sogar nicht-lineare Operatoren. iSAT arbeitet also mit sog. *arithmetischen Transitionssystemen*. Während die Korrektheit der k-Induktion davon unbeeinflusst bleibt (der Beweis aus Abschnitt 4.2 ist unabhängig von der Natur des zugrundeliegenden Systems), ist dies für die Vollständigkeit der Methode leider nicht der Fall. Die in Abschnitt 4.3 geführte Argumentation für die Vollständigkeit von Ansatz 3 für boolsch kodierte Transitionssysteme beruft sich auf die Tatsache, dass solche Systeme nur endlich viele Zustände haben können. Sobald ein System aber reellwertige Variablen enthält, wird die Anzahl der Zustände potentiell *unendlich*, da jede dieser Variablen selbst in einem beliebig kleinen Intervall bereits unendlich viele Werte annehmen kann. Im Allgemeinen ist die k-Induktion für arithmetische Transitionssysteme deshalb nicht vollständig. In einigen Fällen ist sie aber dennoch ausreichend, um die Korrektheit eines solchen Systems nachzuweisen, wie verschiedene Tests nachweisen konnten. Diese Tests werden im Detail in Kapitel 6 diskutiert.

Nun soll die Implementierung der k-Induktion in iSAT beschrieben werden.

5 Implementierung

Die k-Induktion wurde in drei Schritten in den in Kapitel 3 vorgestellten SMT-Solver iSAT implementiert. Dadurch verfügt iSAT über drei k-Induktions-*Modi*, deren Vor- und Nachteile in Kapitel 6.2 diskutiert werden. Im Einzelnen

handelt es sich bei den drei Implementierungen um die *naive* Implementierung, die Ansatz 2 aus Kapitel 4 entspricht, eine Implementierung von Ansatz 3 mit *teurer* Uniqueness, und eine Implementierung mit *dynamischer* Uniqueness. Im Folgenden sollen die drei implementierten Ansätze vorgestellt werden.

5.1 Naive Implementierung

Diese erste Implementierung der k-Induktion in iSAT verzichtet auf das Hinzufügen der Formeln vom Typ $\mathbf{Unique}_k$ und ist somit auch für rein boolsche Transitionssysteme nicht vollständig. Wie Abschnitt 6.2 jedoch zeigen wird, können schon mit diesem naiven Ansatz viele Probleminstanzen gelöst werden. Die Funktionen **inductionBaseHold**(n) und **inductionStepHolds**(n) enthalten jeweils einen Aufruf des Solvers. Dabei liefern sie den Rückgabewert *true*, wenn die zu lösende Formel unerfüllbar ist, und *false*, falls eine erfüllende Belegung gefunden wurde. **inductionBaseHold**(n) prüft die Formel $\mathbf{Base}_n$ und **inductionStepHolds**(n) prüft die Formel $\mathbf{Step}_{n-1}$[6]. Um dabei bestmöglich vom inkrementellen Lösen profitieren zu können, werden neben den Formeln vom Typ $\neg P(s_k)$ auch die $I(s_0)$-Anteile der Formel als *entfernbar* markiert, so dass der Solver im Induktionsschritt von den im Induktionsanfang erlernten Klauseln profitieren kann (siehe Abschnitt 3.4). Algorithmus 2 zeigt diese naive Implementierung. Bis auf das Fehlen der Uniqueness-Klauseln entspricht er genau dem in Abschnitt 4.3 vorgestellten Algorithmus. In Zeile 1 wird zunächst $\mathbf{Base}_0$ überprüft. Ist diese Formel erfüllbar, so meldet der Solver, dass eine mögliche Lösung gefunden wurde, eine sog. *Candidate Solution* (siehe Abschnitt 3.3). Ansonsten wird als nächstes in Zeile 5 $\mathbf{Step}_0$ überprüft. Ist dieser unerfüllbar, so ist das System sicher (*Safe*) bezüglich der Eigenschaft P. Ist dies nicht der Fall, so werden im Folgenden abwechselnd $\mathbf{Base}_k$ (Zeile 8) und $\mathbf{Step}_k$ (Zeile 5) überprüft, wobei k in jedem Schritt erhöht wird, bis entweder ein Ergebnis vorliegt, oder k einen vorher definierten Höchstwert *max_depth* erreicht, in welchem Fall das Ergebnis unbekannt (*Unknown*) bleibt. Dies geschieht beispielsweise dann, wenn das System (wie in Abschnitt 4.3 beschrieben) *Schleifen* aufweist, da die naive Implementierung wie bereits erwähnt nicht vollständig ist.

[6] An dieser Stelle unterscheidet sich die Implementierung geringfügig von dem in Algorithmus 1 vorgestellten Verfahren. Da aber nach wie vor für jede Tiefe k erst $\mathbf{Base}_k$ und anschließend $\mathbf{Step}_k$ geprüft wird, ändert sich nichts an der in Abschnitt 5.1 bewiesenen Korrektheit des Verfahrens.

Algorithm 2 Naive Implementierung

```
 1: if !inductionBaseHolds(0) then
 2:   return CANDIDATE_SOLUTION
 3: end if
 4: for k=1; k ≤ max_depth; k++ do
 5:   if inductionStepHolds(k) then
 6:     return SAFE
 7:   else
 8:     if !inductionBaseHolds(k) then
 9:       return CANDIDATE_SOLUTION
10:     end if
11:   end if
12: end for
13: return UNKNOWN
```

5.2 Implementierung mit teurer Uniqueness

Um Vollständigkeit für alle boolschen Probleminstanzen zu erreichen, wurde der Algorithmus 2 um teure Uniqueness (wie in Abschnitt 4.3 vorgestellt) erweitert. Die entsprechende Implementierung zeigt Algorithmus 3. Den einzigen Unterschied zu Algorithmus 2 stellt die in Zeile 5 aufgerufene Funktion **addUniqueness**(n) dar, die die Teilformel $\mathbf{Unique}_{n-1}$ zur aktuellen Formel hinzufügt. Zu diesem Zweck stellt iSAT eine Funktion **addSeperateFormula**(i,j) zur Verfügung. Sind i und j verschieden, fügt diese eine Formel zur KNF hinzu, die verlangt, dass die Zustände im i-ten und j-ten Schritt einer Lösung unterschiedlich sein müssen. Dabei gilt der Zustand genau dann als unterschiedlich, wenn mindestens eine der Latch-Variablen in Schritt i einen anderen Zustand annimmt als in Schritt j (siehe Abschnitt 4.3). Die Funktion **addUniqueness**(n) ruft $n-1$-mal die Funktion **addSeperateFormula**(i,n) auf, um sicherzustellen, dass alle vorangehenden Zustände sich vom aktuellen Zustand n unterscheiden. Da sich für jede Lösung der Formel $\mathbf{Step}_{k-1}$ der letzte Zustand s_k von all seinen Vorgängern unterscheidet, da er als einziger $\neg P$ erfüllt, reicht es zur Vollständigkeit der Formel aus, zu fordern, dass die k ersten Zustände verschieden sind. Deshalb kann **addUniqueness**(k) alternativ auch erst nach Zeile 12 ausgeführt werden. Da so in jedem Schritt weniger Klauseln betrachtet werden müssen, wird die Laufzeit des Algorithmus unter Umständen verkürzt, ohne dass Korrektheit und Vollständigkeit darunter leiden. Beide Ansätze werden in Abschnitt 6.2 verglichen und diskutiert.

Algorithm 3 Teure Uniqueness

```
1: if !inductionBaseHolds(0) then
2:   return CANDIDATE_SOLUTION
3: end if
4: for k=1; k ≤ max_depth; k++ do
5:   addUniqueness(k)
6:   if inductionStepHolds(k) then
7:     return SAFE
8:   else
9:     if !inductionBaseHolds(k) then
10:      return CANDIDATE_SOLUTION
11:    end if
12:  end if
13: end for
14: return UNKNOWN
```

5.3 Dynamische Uniqueness

Algorithmus 4 schließlich ersetzt die statische durch dynamische Uniqueness, in der Hoffnung, eine Verbesserung der durchschnittlichen Laufzeit des Algorithmus zu erzielen (siehe Abschnitt 6.2). Der Induktionsschritt in Zeile 9 wird dabei so lange wiederholt ausgeführt, bis er unerfüllbar ist oder die Funktion **addUniquenessDynamic**() *false* zurückgibt. Diese Funktion analysiert die aktuelle Lösung des Solvers und sucht nach identischen Zuständen. Für jedes Paar von identischen Zuständen i und j (mit $i \neq j$) wird **addSeperateFormula**(i,j) aufgerufen. Existiert in der aktuellen Lösung kein solches Paar, so gibt die Funktion *false* zurück. Anschließend wird die derart ergänzte Formel erneut dem Solver übergeben. Hierbei ist allerdings zu beachten, dass iSAT in aller Regel für jede Variable keinen konkreten Wert ausgibt, sondern ein Intervall, das den möglichen Wert der Variable enthält. Für boolsche und ganzzahlige Variablen wird ein Punktintervall angegeben, also ein Intervall, bei dem die Ober- und Untergrenze übereinstimmen. Zwei solche Variablen nehmen also dann den gleichen Wert an, wenn ihre Intervalle übereinstimmen. Für reellwertige Variablen hingegen kann nur in Ausnahmefällen ein Punktintervall bestimmt werden. Um zu überprüfen, ob zwei Zustände bezüglich ihrer reellwertigen Variablen identisch sind, wird an dieser Stelle deshalb die sog. *Intervallgleichheit* eingeführt. Im Rahmen dieser Arbeit sollen zwei Intervalle als gleich gelten, wenn sowohl ihre Ober- als auch ihre Untergrenzen übereinstimmen. Die Funktion **intervalEqu**(I_1,I_2) vergleicht zwei Intervalle. Durch die Uniqueness wird gefordert, dass die Variablen in un-

terschiedlichen Zuständen unterschiedliche Werte annehmen. Selbst wenn die Intervallgrenzen der beiden Variablen übereinstimmen, können die konkreten Variablen innerhalb des Intervalls jedoch noch immmer unterschiedliche Werte annehmen. Deshalb kann auch nach dem Hinzufügen wieder die selbe Lösung gefunden werden. Um zu verhindern, dass in diesem Fall eine Schleife auftritt, in der immer wieder die gleichen Klauseln hinzugefügt werden, muss eine Liste über alle Variablenpaare geführt werden, für die die Uniqueness-Klauseln bereits hinzugefügt wurden. Diese wird von der Funktion **addUniquenessDynamic()** intern verwaltet. Der Nutzen der Uniqueness für reellwertige Variablen wird im Einzelnen in Abschnitt 6.1 diskutiert.

Ein Vergleich der drei vorgestellten Ansätze im Bezug auf ihre Performanz und Vollständigkeit hinsichtlich bestimmter Klassen von Problemen wird im nächsten Kapitel durchgeführt.

Algorithm 4 Dynamische Uniqueness

```
 1: if !inductionBaseHolds(0) then
 2:   return CANDIDATE_SOLUTION
 3: end if
 4: for k=1; k ≤ max_depth; k++ do
 5:   if inductionStepHolds(k) then
 6:     return SAFE
 7:   else
 8:     while addUniquenessDynamic() do
 9:       if inductionStepHolds(k) then
10:         return SAFE
11:       end if
12:     end while
13:     if !inductionBaseHolds(k) then
14:       return CANDIDATE_SOLUTION
15:     end if
16:   end if
17: end for
18: return UNKNOWN
```

6 Ergebnisse

In Kapitel 5 wurden die drei k-Induktions-Modi vorgestellt, die in iSAT implementiert wurden. In diesem Kapitel sollen diese drei Implementierungen

bezüglich ihrer Mächtigkeit (also ihrer Fähigkeit, bestimmte Klassen von Probleminstanzen zu lösen) und ihrer Performanz verglichen werden. Dies geschieht anhand einer Reihe von *Benchmarks*. Als *Benchmarks* bezeichnet man Probleminstanzen, deren Lösung bereits bekannt ist, und die primär zum Test eines entwickelten Lösungsverfahrens verwendet werden.

6.1 Mächtigkeit

Wie bereits in Abschnitt 4.4 erwähnt, ist die k-Induktion nur für bestimmte Arten von Transitionssystemen ein vollständiger Lösungsansatz. In diesem Abschnitt soll anhand von Beispielen dargestellt werden, welche Art von Benchmarks die k-Induktion in iSAT lösen kann und welche nicht.
Es wurde bereits gezeigt, dass die k-Induktion vollständig ist für rein boolsche Systeme. Ein boolscher Benchmark kann also auf jeden Fall gelöst werden, sofern genug Zeit und Speicher zur Verfügung stehen. Ähnlich verhält es sich bei Benchmarks, die ganzzahlige Variablen enthalten: Jede ganze Zahl kann problemlos durch einen Vektor von boolschen Variablen dargestellt werden. Da eine ganzzahlige Variable durch die Intervallbegrenzung nur endlich viele Werte annehmen kann, ist der *Zustandsraum* (die Menge aller in einem System enthaltener Zustände) endlich. Auch für solche Systeme ist die k-Induktion deshalb vollständig.
Problematisch wird es erst in Bezug auf arithmetische Transitionssysteme, die reellwertige Variablen enthalten. Und zwar in zweierlei Hinsicht: Wie bereits erwähnt, haben solche Systeme einen potentiell unendlichen Zustandsraum. Somit reichen die Uniqueness-Klauseln im Allgemeinen nicht mehr aus, um die Vollständigkeit der k-Induktion zu gewährleisten. Und insbesondere iSAT, der die Uniqueness-Klauseln für reellwertige Variablen durch die Intervallarithmetik nicht mehr richtig nutzen kann[7], kann hier viele Eigenschaften nicht mehr verifizieren, obwohl sie im System gültig sind.
Ein Beispiel hierfür ist die sog. *Gingerbreadman Map* [14]. Es handelt sich dabei um eine *chaotische Map*. Eine *Map* ist eine Abbildung von einer oder mehreren Variablen auf Nachfolgewerte. Ausgehend von beliebige ausgewählten Startwerten erhält man eine Folge von Punkten, die in ein Koordinatensystem eingetragen werden können[8]. Oft bilden die so entstandenen Linien bestimmte Muster. Für chaotische Maps sind die Punkte im Allgemeinen jedoch scheinbar zufällig angeordnet.

Beispiel 4 (Gingerbreadman Map)**.** Die Gingerbreadman Map wird durch die Übergangsfunktion für zwei Werte x und y definiert.

[7] Siehe Abschnitt 5.3.
[8] Zumindest dann, wenn zwei oder drei Variablen betrachtet werden.

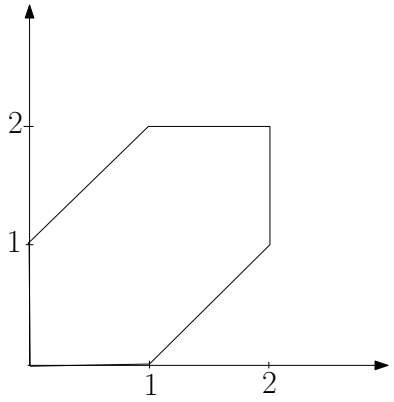

Abbildung 6: Periodisches Verhalten der Gingerbreadman Map für $x \in [0, 3]$ und $y \in [0, 3]$. Für alle Punkte in dem gezeichneten Sechseck hat die Funktion eine Periode von 6, mit Ausnahme des Mittelpunktes (dort ist die Periode 1). Punkte außerhalb des Secksecks verhalten sich aperiodisch und chaotisch.

- $x_{n+1} = 1 - y_n + |x_n|$
- $y_{n+1} = x_n$

Im Allgemeinen ist das Verhalten der Gingerbreadman Map chaotisch und aperiodisch. Für bestimmte Bereiche jedoch (wie zum Beispiel das Sechseck in Abbildung 6.1) wird das Verhalten *periodisch*. Jeder Punkt in diesem Sechseck hat die Periode 6 (mit Ausnahme des Mittelpunkts (1,1), der eine Periode von 1 hat), nach 6 Schritten wird also wieder der Ausgangswert erreicht. Alle Punkte dazwischen liegen ebenfalls innerhalb des Sechsecks. Folglich gilt die Eigenschaft, dass, ausgehend von einem Wert innerhalb des Sechsecks, niemals ein Wert erreicht werden kann, der außerhalb davon liegt. Doch obwohl sich die Werte nach spätestens 6 Schritten wiederholen, kann diese Eigenschaft mithilfe der k-Induktion nicht verifiziert werden, da die Uniqueness-Klauseln aufgrund der Intervalle für reellwertigen Variablen nicht greifen. Da die Gingerbreadman Map nur lineare Operationen enthält, kann der Benchmark auch mit herkömmlichen SMT-Solvern gelöst werden, bespielsweise mit dem in Abschnitt 3.1 vorgestellten Lazy Theorem Proving. Da diese in der Regel nicht mit Intervallen rechen, sondern nach exakten Lösungen suchen, würden hier vermutlich die Uniqueness-Formeln greifen. Diese Solver wären also wahrscheinlich in der Lage, die gesuchte Eigenschaft mithilfe der k-Induktion zu beweisen.

Unter bestimmten Umständen kann die Korrektheit einer Eigenschaft jedoch dennoch nachgewiesen werden, beispielsweise dann, wenn ein System

keine Schleifen enthält und die Uniqueness-Klauseln überhaupt nicht benötigt werden, oder wenn nur solche Zustände erreichbar sind, in denen die Werte der Variablen durch Punktintervalle angegeben werden können.
Außerdem kann unter gewissen Umständen die zu überprüfende Eigenschaft so modifiziert werden, dass eine Überprüfung wieder möglich ist. Ein Beispiel hierfür ist die sog. *Gauss Map* [15]. Anders als bei der Gingerbreadman Map wird hier nur eine Variable betrachtet. Das Verhalten der Funktion hängt hier nicht mehr vom Startwert der Variable ab, sondern von zwei Paramtern α und β.

Beispiel 5 (Gauss Map)**.** Die Gauss Map wird durch die Übergangsfunktion für eine Variable x definiert.

- $x_{n+1} = exp(-\alpha x_n^2) + \beta$

Für beispielsweise $\alpha = 4.9$ und $\beta = -0.1$ nimmt x nach einer gewissen Zeit (der ‚Einpendelphase') abwechselnd genau zwei verschiedene Werte an. Nun soll mithilfe der k-Induktion bewiesen werden, dass nach einer Einpendelphase von 30 Schritten keine anderen Werte mehr erreicht werden können. Da durch die Intervallarithmetik aber immer ein Lösungsintervall statt einer punktgenauen Lösung errechnet wird, kann diese Eigenschaft nicht verifiziert werden. Erweitert man jedoch die Eigenschaft dergestalt, dass auch hier Werte innerhalb eines Zielintervalls betrachtet werden, so kann – bei ausreichender Breite dieses Intervalls – verifiziert werden, dass nach der Einpendelphase keine anderen Werte mehr angenommen werden.
Ob und wie eine Eigenschaft verändert werden kann, hängt vom betrachteten System und der Spezifikation ab und muss individuell entschieden werden. In vielen Fällen ist es ausreichend, nachzuweisen, dass sich Werte innerhalb eines bestimmten Intervalls befinden. Wenn es für die Korrektheit eines System jedoch entscheidend ist, einen exakten Wert zu erreichen, darf die Eigenschaft nicht auf diese Art und Weise modifiziert werden. Es ist dann jedoch nicht möglich, sie mithilfe von iSAT und der k-Induktion zu verifizieren.

6.2 Performanz

Jede der in Kapitel 5 vorgestellten Implementierungen bietet gewisse Vorteile gegenüber den anderen. Um aufzuzeigen, wo die Stärken und Schwächen der einzelnen Ansätze liegen, wurde eine große Menge von rein boolschen Benchmarks [16] überprüft. Getestet wurden 332 erfüllbare und 460 unerfüllbare BMC-Probleme. Da die naive Implementierung nicht vollständig ist und möglicherweise nicht bei allen Benchmarks terminiert, und die Laufzeit auch für die vollständigen Implementierungen unter Umständen sehr hoch

	Benchmarks gelöst	ØTiefe	ØZeit	ØUnique
ohne Induktion	269	9.3	228.3	0
naive Implementierung	252	7.2	282	0
teure Uniqueness	244	5.9	298.7	48.8
dynamische Uniqueness	247	6	294.7	4.3

Tabelle 1: Performanz der verschiedenen Implementierungen bei erfüllbaren Benchmarks

sein kann, wurde eine maximale Laufzeit von 900 Sekunden pro Benchmark festgelegt. Für Benchmarks, die innerhalb dieser Zeit nicht gelöst werden konnten, wurde das Ergebnis *UNKOWN* (unbekannt) ausgegeben. Das System, auf dem die Tests durchgeführt wurden, verfügt über eine Quad-Core AMD Opteron^{TM}8356 CPU sowie ausreichend Hauptspeicher[9]. Die Tabellen 1 und 2 zeigen die Ergebnisse der Testläufe. Dabei gibt die Spalte *Benchmarks gelöst* an, wieviele der Benchmarks innerhalb der maximalen Laufzeit gelöst werden konnten. *ØTiefe* gibt an, bis zu welcher *Abrolltiefe* k im Durchschnitt erhöht werden musste, und *ØZeit* beschreibt die durchschnittliche Laufzeit jedes Benchmarks. Benchmark, die innerhalb des Zeitlimits nicht gelöst werden konnten, wurden dabei mit 900 Sekunden veranschlagt. *ØUnique* schließlich gibt an, für wieviele Paare von Zuständen s_i und s_j die Uniqueness-Formel $U(s_i, s_j)$ hinzugefügt wurde. Für die erfüllbaren Benchmarks wurde zusätzlich noch ein Vergleich mit iSAT ohne k-Induktion angestellt.

Erfüllbare boolsche Benchmarks

Bei der Suche nach Gegenbeispielen bei den erfüllbaren Benchmarks erzielt iSAT ohne k-Induktion die besten Ergebnisse – es wurden deutlich mehr Benchmarks gelöst als mit den drei k-Induktions-Implementierungen, und auch der durchschnittliche Zeitbedarf pro Benchmark fällt wesentlich geringer aus. Dieses Ergebniss war zu erwarten, denn während der herkömmliche BMC-Ansatz nur die Formeln vom Typ $\mathbf{Base}_k$ prüft, kommt bei den drei Induktions-Implementierungen in jedem Schritt noch eine Formel vom Typ $\mathbf{Step}_k$ hinzu. Da diese sich jedoch nicht nur auf Pfade beschränkt, die von einem Initialzustand ausgehen, ist der Suchraum deutlich größer als bei $\mathbf{Base}_k$. Dadurch wird es zwar schwerer[10], die Unerfüllbarkeit einer unerfüllbaren Formel zu beweisen, aber es ist im Allgemeinen auch leichter, eine erfüllende Belegung für eine erfüllbare Formel zu finden. So lässt sich auch erklären,

[9] iSAT hat eine 32 Bit-Architektur und kann somit nicht mehr als 4GB RAM adressieren
[10] Schwerer bedeutet hier, dass die Laufzeit höher ist.

	Benchmarks gelöst	ØTiefe	ØZeit	ØUnique
naive Implementierung	156	24.5	600.6	0
teure Uniqueness	160	12.3	592.8	204.3
dynamische Uniqueness	161	6	592.9	16.9

Tabelle 2: Performanz der verschiedenen Implementierungen bei unerfüllbaren Benchmarks

wieso die Implementierungen der k-Induktion nicht doppelt so lange benötigen, um ein Gegenbeispiel zu finden, sondern lediglich ca. 1.25-mal so lange.
Die Uniqueness-Klauseln, die den Induktionsschritt verstärken sollen, um das Verfahren für boolsche Benchmarks vollständig zu machen, werden bei der Suche nach Gegenbeispielen nicht benötigt, da hier lediglich die Lösung von $\mathbf{Base}_k$ von Bedeutung ist. Durch die hinzugefügten unnötigen Klauseln wird allerdings der Solver zusätzlich belastet. Folglich zeigen die Implementierungen mit Uniqueness hier ein etwas ungünstigeres Laufzeitverhalten als die naive Implementierung. Es ist jedoch bereits zu erkennen, dass die Implementierung mir dynamischer Uniqueness deutlich weniger Klauseln hinzufügt als die Implementierung mit teurer Uniqueness – nur etwa ein Zwölftel der Klauseln wurde tatsächlich benötigt.

Unerfüllbare boolsche Benchmarks

Bei den unerfüllbaren Benchmarks sieht man deutlich, dass einige Benchmarks nur unter Zuhilfenahme von Uniqueness-Klauseln gelöst werden konnten. Die naive Implementierung löste weniger Benchmarks als die Implementierungen mit Uniqueness. Ein Beispiel hierfür ist der Benchmark *pdtvisgray1*. Die naive Implementierung erreichte das Timeout von 900 Sekunden bei einer Tiefe von $k = 2873$, während die Implementierung mit teurer Uniqueness schon nach 0.04 Sekunden bei einer Tiefe von $k = 8$ terminierte. Dadurch erklärt sich auch, wieso die durchschnittliche Suchtiefe der naiven Implementierung beinahe doppelt so hoch ist wie die der anderen Implementierungen. Aber auch bei Benchmarks, die keine der drei Implementierungen innerhalb des Zeitlimits lösen konnte, erreichte die naive Implementierung meist eine größere Tiefe, da in jedem Solver-Lauf deutlich weniger Klauseln betrachtet werden mussten. Dies zeigt deutlich z.B. der Benchmark *kenflashp06*. Hier erreichte die naive Implementierung eine Tiefe von 149, während die Ansätze mit Uniqueness schon bei $k = 38$ ihren Timeout erreichten. Für Benchmarks, zu deren Lösung Uniqueness-Klauseln benötigt werden, hat die Implementierung

SAT	Benchmarks gelöst	ØTiefe	ØZeit	ØUnique
teure Uniqueness(1)	244	5.9	298.7	48.8
teure Uniqueness(2)	246	6	294.8	43.8
UNSAT	Benchmarks gelöst	ØTiefe	ØZeit	ØUnique
teure Uniqueness(1)	160	12.3	592.8	204.3
teure Uniqueness(2)	160	12.4	592.9	198.4

Tabelle 3: Vergleich der zwei vorgestellten Varianten der teuren Uniqueness

mit dynamischer Uniqueness stets etwas höhere Laufzeiten zu verzeichnen als jene mit teurer Uniqueness. Für den Benchmark *bobcount* beispielsweise wurden 6.33 Sekunden mit dynamischer und nur 5.34 Sekunden mit teurer Uniqueness benötigt. Dies erklärt sich dadurch, dass bei der dynamischen Implementierung der Uniqueness die Formel $\mathbf{Step}_k$ für jedes k unter Umständen mehrmals gelöst werden muss - nämlich so oft, bis eine Lösung gefunden wurde, die keine Paare von identischen Zuständen mehr enthält. Dafür werden insgesamt deutlich weniger Uniqueness-Klauseln benötigt – bei *bobcount* sind es nur noch 24 statt 171, im Schnitt wird nur der zwölfte Teil der teuren Uniqueness-Klauseln benötigt. Dadurch wird während der Ausführung des Solvers deutlich weniger Speicher belegt. Da iSAT durch seine 32-Bit-Architektur maximal 4GB Hauptspeicher adressieren kann, kann diese Speicherersparnis in praktischen Anwendungen ein wichtiger Faktor dafür sein, ob ein Ergebnis berechnet werden kann oder nicht. Im Rahmen der hier durchgeführten Tests trat jedoch kein Fall auf, in dem durch Hinzufügen von Uniqueness-Klauseln zu viel Speicher benötigt wurde.

Varianten der teuren Uniqueness

Tabelle 3 vergleicht die beiden in Abschnitt 5.2 vorgestellten Implementierungen der teuren Uniqueness. Bei Implementierung (1) wird **addUniqueness**(k) vor **inductionStepHolds**(k) aufgerufen, bei Implementierung (2) erst vor **inductionStepHolds**(k+1). Wie man sieht, ist die zweite Implementierung etwas besser, wobei allerdings die Unterschiede in Laufzeit und erreichter Suchtiefe gering sind. Allerdings war schon diese kleine Verbesserung ausreichend, um 246 statt nur 244 der erfüllbaren Benchmarks zu lösen. Es ist also empfehlenswert, in der Praxis die zweite Implementierung einzusetzten.

SAT	Benchmarks gelöst	ØTiefe	ØZeit
ohne k-Induktion	36	28.8	208.4
naive Implementierung	25	20.7	371.4
UNSAT	Benchmarks gelöst	ØTiefe	ØZeit
naive Implementierung	4	12	8.4

Tabelle 4: Performanz für nicht-lineare Benchmarks

Nicht-lineare Benchmarks

In Abschnitt 6.1 wurde bereits diskutiert, dass die k-Induktion in iSAT für Benchmarks mit reellwertigen Variablen und somit insbesondere auch für solche mit nicht-linearen, transzendenten Funktionen im allgemeinen nicht vollständig ist. Dennoch können unter Umständen auch bei diesen Benchmarks Ergebnisse erzielt werden. Die Entwickler von iSAT stellen einige solcher Benchmarks zur Verfügung (43 erfüllbare und 6 unerfüllbare), die zu diesem Zweck analysiert wurden. Zusätzlich wurde auch die in Abschnitt 6.1 vorgestellten Gauss Map getestet. Da bereits gezeigt wurde, dass das Hinzufügen von Uniqueness-Klauseln bei dieser Art von Benchmarks keinen Vorteil bringt, werden hier nur die Ergebnisse der naiven Implementierung betrachtet. Die Ergebnisse zeigt Tabelle 4.
Für erfüllbare Benchmarks ist das Verhältnis der Performanz ohne k-Induktion und mit der naiven Implementierung ähnlich wie für die boolschen Benchmarks. Bei den unerfüllbaren Benchmarks konnte die Unerfüllbarkeit immerhin bei 4 von 7 Benchmarks nachgewiesen werden. Im Gegensatz zu den Tests mit boolschen Benchmarks wurde in den verbleibenden 3 Fällen allerdings nicht die maximale Laufzeit von 900 Sekunden überschritten, sondern eine Candidate Solution gefunden. Dies ist jedoch kein Fehler, sondern liegt daran, dass iSAT mit seinen Standardeinstellungen solange rechnet, bis ein Lösungsintervall der Breite 0.1 gefunden wurde. Wie bereits in Abschnitt 3.3 dargestellt wurde, muss dieses Intervall allerdings nicht zwangsläufig eine Lösung der Formel enthalten. Wiederholt man die Tests mit einer maximalen Intervallbreite von 0.001, so sind auch die verbleibenden 3 Benchmarks lösbar, wobei allerdings die benötigte Laufzeit und der Speicherbedarf stark ansteigen. Der vorher lösbare Benchmark *gasburner* kann mit der höheren Genauigkeit deshalb nicht mehr gelöst werden, da hier der Speicherbedarf den maximal adressierbaren Hauptspeicher übersteigt.

In diesem Kapitel wurde gezeigt, wie sich die unterschiedlichen Implementierungen der k-Induktion in iSAT beim Lösen von boolschen und nicht-boolschen

Benchmarks verhalten. Das nächste Kapitel bietet eine Zusammenfassung der im Laufe dieser Arbeit gemachten Beobachtungen und einen Ausblick auf mögliche Erweiterungen, durch die die k-Induktion in iSAT noch effizienter werden könnten.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die k-Induktion vorgestellt. Ihre Korrektheit wurde bewiesen, und es wurde gezeigt, wie sie mit Hilfe von verschiedenen Mechanismen vollständiger und effizienter gemacht werden kann. Anschließend wurde die k-Induktion für den BMC-Solver iSAT implementiert. Dabei wurden insgesamt drei verschiedene Implementierungen betrachtet, die unterschiedlich effizient arbeiten und von denen zwei für boolsche Benchmarks vollständig sind. In Abschnitt 6.1 wurde außerdem eine Möglichkeit umrissen, die zu verifizierende Eigenschaft so zu verändern, dass auch für viele Systeme mit reellwertigen Variablen eine Verifikation möglich ist. In Abschnitt 6.2 wurde die Performanz der verschiedenen Implementierungen anhand verschiedener Klassen von Benchmarks verglichen. Es wurde gezeigt, dass die Unerfüllbarkeit sowohl boolscher als auch vieler nicht-linearer Systeme mit der k-Induktion in iSAT nachgewiesen werden kann.
Während es vermutlich nicht möglich sein wird, auf Grundlage der k-Induktion in iSAT eine vollständige Methode zur Verifikation beliebiger Eigeschaften zu entwickeln, gibt es dennoch Ideen, mit deren Hilfe mehr Eigenschaften verifiziert werden können.
Während reellwertige Variablen unendlich viele Werte annehmen können, kann man in realen Systemen diese Werte oft nur bis zu einem gewissen Grad unterscheiden. Dadurch kann der Zustandsraum eingeschränkt werden. Enthält ein System beispielsweise ein Thermometer, das die Temperatur bis auf drei Nachkommastellen genau misst, dann müssen auch im Modell keine Zustände betrachtet werden, die zwischen zwei real messbaren Werten liegen. Es wäre also möglich, die Uniqueness-Klauseln so zu modifizieren, dass Variablen schon dann als gleich betrachtet werden, wenn sie sich ausreichend *ähnlich* sind. In dem so erzeugten endlichen Zustandsraum wäre die k-Induktion wiederum vollständig.
Außerdem wäre es eventuell möglich, die allgemeine Performanz des Verfahrens zu verbessern, indem statt nur einer Instanz von iSAT mehrere Instanzen parallel verwendet werden, um Induktionsbasis und -schritt zu lösen. Solche parallelen Ansätze wurden in vielen BMC-Prozeduren bereits erfolgreich implementiert und könnten unter Umständen auch die Laufzeit der k-Induktion deutlich verbessern.

Literatur

[1] Eén, N., Sörensson, N.: Temporal induction by incremental sat solving. Electr. Notes Theor. Comput. Sci. **89**(4) (2003) 543–560

[2] Baier, C., Katoen, J.P.: Principles of model checking (2008)

[3] Kreuzer, M., Kühling, S.: Logik für Informatiker. Pearson (2006)

[4] Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic **2** (1968) 115–125

[5] Molitor, P., Becker, B.: Technische Informatik. Oldenbourg Wissenschaftsverlag (2008)

[6] Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing. STOC '71, New York, NY, USA, ACM (1971) 151–158

[7] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5** (1962) 394–397

[8] Schubert, T.: SAT-Algorithmen und Systemaspekte: vom Mikroprozessor zum parallelen System. PhD thesis, Universität Freiburg (2008)

[9] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19** (2001) 7–34

[10] AVACS: Automatic verification and analysis of complex systems (2010) http://www.avacs.org.

[11] Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling, and Computation **1** (2007) 209–236

[12] Wimmer, R.: Vorlesung: Verifikation probabilistischer und hybrider systeme (2012)

[13] iSAT Developer Team: iSAT Quickstart Guide. (2010)

[14] L., R., Devaney: A piecewise linear model for the zones of instability of an area-preserving map. Physica D: Nonlinear Phenomena **10**(3) (1984) 387–393

[15] Hilborn, R.C.: Chaos and Nonlinear Dynamics: An Introduction for Scientists and Engineers. 2. edn. Oxford University Press (2001)

[16] Hardware Model Checking Competition. (2010) http://fmv.jku.at/hwmcc10/.